Automated Bug Report Field Reassignment and Refinement Prediction

Xin Xia, *Member, IEEE,* David Lo, *Member, IEEE,* Emad Shihab, Xinyu Wang

***Abstract*–Bug fixing is one of the most important activities in software development and maintenance. Bugs are reported, recorded, and managed in bug tracking systems such as Bugzilla. In general, a bug report contains many fields, such as product, component, severity, priority, fixer, operating system (OS), platform, etc., which provide important information for the bug triaging and fixing process. Our previous study finds that approximately 80% of bug reports have their fields reassigned and refined at least once, and bugs with reassigned and refined fields take more time to fix than bugs with no reassigned and refined fields. Thus, automatically predicting which bug report fields get reassigned and refined could help developers to save bug fixing time.**

Considering that a bug report could have multiple field reassignments and refinements (e.g., the product, component, fixer, and other fields of a bug report can get reassigned and refined), in this paper, we propose a multi-label learning algorithm to predict which bug report fields might be reassigned and refined. We note that the number of bug reports with some types of reassignment and refinement (e.g., bugs whose severity fields gets reassigned and refined) is a small proportion of the whole bug report collection, indicating the *class imbalance* problem. Thus, we propose imbalanced ML.KNN (*Im-ML.KNN*), which extends ML.KNN, one of the state-of-the-art multi-label learning algorithms, to achieve better performance. *Im-ML.KNN* is a composite model that combines 3 multi-label classifiers built using different types of features (i.e., meta, textual, and mixed features). We evaluate our solution on 4 large bug report datasets including OpenOffice, Netbeans, Eclipse, and Mozilla containing a total of 190,558 bug reports. We show that *Im-ML.KNN* can achieve an average F-measure score of 0.56-0.62. We also compare *Im-ML.KNN* with other state-of-art methods, such as the method proposed by Lamkanfi *et al.*, ML.KNN, and HOMER-NB. The results show that *Im-ML.KNN*, on average, improves the average F-measure scores of Lamkanfi et al.'s method, *ML.KNN*, and HOMER-NB by 119.69%, 9.11%, and 161.08%, respectively.

***Index Terms*—Bug Report Field Reassignment and Refinement, Multi-label Learning, Imbalance Learning, Composite Model**

ACRONYMS AND ABBREVIATIONS

BRFRR	Bug Report Field Reassignment and Refinement
Im-ML.KNN	Imbalanced ML.KNN
NB	Naive Bayes
KNN	K-nearest Neighbors
LDS	Longitudinal Data Setup

Xin Xia and Xinyu Wang are with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. (E-mail: xxia@zju.edu.cn, wangxinyu@zju.edu.cn)

David Lo is with the School of Information Systems, Singapore Management University, Singapore. (E-mail: davidlo@smu.edu.sg)

Emad Shihab is with the Department of Computer Science and Software Engineering, Concordia University, Montreal, QC, Canada. (E-mail: eshihab@cse.concordia.ca)

Xinyu Wang is the corresponding author.

NOTATION

BR	A set of bug report collection
b_i	The ith bug report
$w_{i,j}$	Weight for j^{th} term in b_i
L	A set of labels
$threshold_l$	Threshold for label l
P_l	Precision for label l
R_l	Recall for label l
F_l	Precision for label l
$Ave.P$	Average Precision for labels in L
$Ave.R$	Average Recall for labels in L
$Ave.F$	Average F-measure for labels in L
$Sample$	Sample size
$Meta$	Meta features
$Text$	Textual features
$Mixed$	Mixed features

I. INTRODUCTION

Bugs are inevitable in the whole lifecycle of software development and maintenance, and bug fixing is a time-consuming and costly task. Previous studies from NIST show that software bugs cost the US economy an estimated $59 billion every year, which is around 0.6% of the gross domestic product [1]. Bug tracking systems such as Bugzilla are used to report, record, and manage these bugs. A typical bug report contains many fields, e.g., the summary and description fields which provide the textual description of the observed bug, the status field which shows the current status (e.g., *closed* or *resolved*), the product and component fields where the bug is detected, the priority and severity fields which mark the importance of the bug, the version, operating system (OS), and platform fields which indicate the runtime environment affected by the bug, and the reporter and fixer fields. These fields are vital for developers to triage and fix the bug [2], [3].

However, Bug report fields often get reassigned and refined. Various types of bug report field reassignments and refinements have been investigated in the literature. Shihab et al. study reopened bugs, and they find that, for Eclipse Platform 3.0, the average time to resolve a reopened bug is more than twice the time to resolve a non-reopened bug [4], [5]. Lamkanfi and Demeyer studied component reassignments and found that, for Mozilla, it takes a long time to reassign a bug report to the correct component [6]. Jeong et al. studied fixer reassignments and refinements in Eclipse and Mozilla, and found that 37-44% of bugs have their fixer reassigned and refined [7]. Saha et al. found that 10% of long lived bugs get their fixers reassigned and refined 5 or more times [8]. In our previous work, we analyzed bug reports from OpenOffice,

Netbeans, Eclipse, and Mozilla, and found that, approximately 80% of bug reports have their fields reassigned [9]. Also, we found that bug reports with field reassignments have statistically significantly longer bug fix time than those without reassignments.

The aforementioned studies indicate that bug reassignments and refinements are associated to longer bug fixing times. Some fields are wrongly assigned and this can result in a delay for a bug to get resolved, while other fields are not inherently wrong, but need to be adjusted with additional insights that bug triagers have after they analyze the bugs [10]. For these cases, there is a need for an approach that can help developers reduce the amount of incorrect assignments, or to suggest appropriate refinements that developers can consider to make in the future. Such an approach can reduce the number of unnecessary reassignments and refinements. Admittedly, since many factors affect bug fixing time (e.g., difficulty to reproduce and resolve a bug [8], etc.), such an approach is not a panacea to make bug fixing time short. Nevertheless, it helps solve a problem that impacts bug fix time.

To address the above-mentioned need, in this paper, we develop a tool that leverages *multi-label learning* algorithms to automatically predict which *bug report fields will be reassigned or refined*. In the multi-label learning literature, one data instance (i.e., a bug report) can be assigned to multiple labels (i.e., fields that are predicted to be reassigned and refined) [11]. It is important to note that our work complements previous studies such as the work on reopened bug prediction [4], [5] and component reassignment and refinement prediction [6], [12], since our work generalizes these studies by considering the reassignments and refinements of many different fields in a bug report. Our proposed multi-label learning algorithm can predict, not only the reassignment and refinement of the status field or the component field, but also all other fields simultaneously[1]. To investigate the usefulness of our tool, we have checked with several experienced developers from OpenOffice and NetBeans Project Management Committee, who have fixed hundreds of bugs and managed the bug reports in OpenOffice and Netbeans. Some of their comments are as follows:

"Considering a lot of "raw" users would submit bug reports in our community, there would be many errors (wrongly assigned fields in the bug report), the tool would be possible to evaluate a "raw" user submitted report and predict what fields will be changed."

"*Although human thought was necessary here to decide what the right component (fields) should be (during bug fixing process), a tool which assists whether a fields would get reassigned and refined still relief the workload for a developer.*"

"*I think, a reassignment prediction can be useful, especially when non-developers create bug requests that are not familiar with the development process and management. Such users may fill out some fields incorrectly, which could be detected more easily and help the developers to better assess and organize the reports.*"

[1] In the machine learning literature, the problems of reopened bug prediction and component prediction can be mapped to single-label learning problems. Single-label learning is a classification problem where one instance (e.g., a bug report) can only be assigned to one label (e.g., reopened or not reopened, and reassigned and refined or not reassigned and refined.)

To build our tool, we extract the values of important features from the bug reports when they are initially submitted. The features extracted from a training set of bug reports, along with field reassignment and refinement information, are then used to build a multi-label classifier. We extract field reassignment and refinement information by analyzing the history of bug reports to identify fields that are changed after the bug report was initially submitted. The resultant multi-label classifier serves as a tool and will be used to predict the fields which would get reassigned and refined for a new submitted bug report. The output of our tool is a list of bug report fields which would get reassigned and refined. With our tool, developers will be better informed on whether they have assigned the right field values when they submit a bug report.

To assist in making accurate predictions, one possible solution is to use ML.KNN [13], one of the state-of-the-art algorithms used to solve the multi-label learning problem. However, we find that for many fields, there is only a very small percentage of bug reports whose fields are reassigned and refined. For example, in Eclipse, only 9.76%, 18.44%, 9.19%, and 8.14% of bug reports have their product, component, severity and status fields reassigned and refined [9]. We refer to this phenomenon as the *class imbalance phenomenon* [14]. To improve the overall performance of *ML.KNN*, we propose imbalanced ML.KNN (*Im-ML.KNN*), which addresses the class imbalance phenomenon experienced in the bug report field reassignment and refinement task. *Im-ML.KNN* is a composite model, which combines 3 multi-label classifiers built using different types of features (i.e., meta, textual, and mixed features). In our paper, meta features refer to the non-textual fields of a bug report, e.g., reporter, assignee, product, component, etc, textual features refer to the proceed terms extracted from the description and summary field, and mixed features refer to the combination of both meta and textual features. *Im-ML.KNN* automatically learns the best threshold value to predict which fields will be reassigned and refined in the training data. By default, we set the number of neighbors K in *Im-ML.KNN* as 10.

In our previous study, we perform an empirical study on bug report field reassignment and refinement [9]. This paper complements our previous work, and our previous work serves as a motivation to this work. In particular, in this paper we propose an automated tool to predict which bug report fields will get reassigned, to help developers reduce bug fixing effort.

We evaluate our *Im-ML.KNN* algorithm on bug report datasets from 4 large open source porjects namely - OpenOffice, Netbeans, Eclipse, and Mozilla, containing a total of 190,558 bug reports. The experiment results show that *Im-ML.KNN* can achieve an average F-measure score between 0.52-0.67. We also compare *Im-ML.KNN* with other state-of-art methods, such as the method proposed by Lamkanfi *et al.*, ML.KNN and HOMER-NB [15]. We address the following research questions:

RQ1 What is the F-measure of *Im-ML.KNN*? How much improvement can it achieve over the method

proposed by Lamkanfi et al. [6], ML.KNN [13], and HOMER-NB [15]?
The results show that Im-ML.KNN, on average, improves the F-measure score of Lamkanfi et al.'s method, ML.KNN, and HOMER-NB by 119.69%, 9.11%, and 161.08%, respectively.

RQ2 **Can the F-measure of Im-ML.KNN outperform those of its constituent components (i.e., meta classifier, text classifier, and mixed classifier)?**
Yes, Im-ML.KNN improves the average F-measure scores of meta classifier, text classifier, and mixed classifier by 8.91%, 164.31%, and 9.11%, respectively. The results show that it is beneficial to combine the 3 classifiers.

RQ3 **Do different numbers of neighbors affect the F-measure of *Im-ML.KNN*?**
No, across the 4 projects, Im-ML.KNN achieves a relatively stable performance when different numbers of neighbors are used.

RQ4 **What are good predictors of bug report field reassignments and refinements? Do the predictors differ for different fields?**
Meta features (e.g., *product*, *component*, *assignee*) make up most of the top-10 features. Among the 4 projects, product, component, reporter, and assignee are the 4 most important meta features related to various types of field reassignment.

RQ5 **What is the effect of varying the amount of training data on the effectiveness of *Im-ML.KNN*?**
To reduce the amount of training data, we perform 10 times K-fold cross-validation, with K varied from 2 to 10. When we vary K from 10 to 2, the F-measures for Eclipse, Mozilla, and Firefox remains relatively stable (it fluctuates less than 5.68% from the original value). For OpenOffice, the F-measure reduces by 26.81% when we vary k from 10 to 2.

RQ6 **How much time does it take for Im-ML.KNN to run?**
The average model building time and the average prediction time of Im-ML.KNN is 0.0265 and 0.0158 seconds per bug report, respectively.

The main contributions of this paper are:

- **Propose a new algorithm that effectively deals with the class imbalance problem.** Considering the class imbalance phenomenon, we propose a new algorithm named imbalanced ML.KNN (*Im-ML.KNN*) to achieve better performance when predicting reassigned and refined fields.
- **Accurately predict which bug fields will be reassigned and refined.** We propose a multi-label learning algorithm to accurately predict which bug fields will be reassigned and refined. To the best of our knowledge, this is the first study to use *multi-label learning* to predict bug report field reassignments and refinements.
- **Perform an extensive empirical study to examine the effectiveness of *Im-ML.KNN* in predicting which bug fields will be reassigned and refined.** We investigate the performance of *Im-ML.KNN* on 4 large open-source projects, and the experiment results show that our method improves existing state-of-the-art methods such as Lamkanfi *et al.*'s method and ML.KNN.

The remainder of the paper is organized as follows. We describe the preliminary materials in Section II. We outline the overall framework of our bug report field reassignment and refinement prediction solution in Section III. We elaborate how the features and labels (i.e., various bug report field reassignments and refinements) are extracted from bug reports in Section IV. We present our multi-label classification approach *Im-ML.KNN* in Section V. We report the experiment results in Section VI. We discuss and present the threats to validity of our paper in Section VII and VIII. We describe related work in Section IX. We conclude and mention future work in Section X.

II. Preliminaries

In this section, we first present the background of bug report field reassignment and refinementin Section II-A. Next, we describe ML.KNN, which is the state-of-the-art multi-label classification algorithm that we build our approach on, in Section II-B.

A. Background

A typical bug report contains many useful fields, such as product, component, fixer, summary, description, etc. However, in some cases, the fields in the bug report get reassigned and refined. Figure 1 shows a bug report from Eclipse with BugID 221068[2]. We notice that the product, component, fixer, and status fields of this bug report have been reassigned and refined. The product was reassigned from `WTP Incubator` to `WTP Source Editing`, and the component was reassigned from `incubator` to `wtp.inc.xsl`, and finally it was reassigned to `wst.xsl`. The fixer was reassigned from `wtp.inc-inbox` to `doug.satchwell`. Moreover, the bug report in Figure 1 was also a reopened bug, i.e., the bug report was first resolved and fixed by `doug.satchwell`, and then `d_a_carver` reopened it and changed the status to new. In this paper, we only consider one type of *status* reassignment: *resolved* or *closed* to *reopen*. This is the one of most important reassignment and refinement. We ignore the other status reassignments and refinements as in general a bug report status would eventually get changed (e.g., from *open* to *closed*) as developers are working to fix it.

Observations and Implications. From the above bug report, we make the following observations:

1) The bug report was created on March 2nd, 2008, and it was fixed on April 30th, 2009. This bug took approximately one year to get fixed.
2) The component was reassigned from `incubator` to `wtp.inc.xsl` on June 5th, 2008, by `d_a_carver`. However, on March 31st, 2009, `webmaster` still reassigned its component and product fields. Thus, it seems that even though 9 months had passed, a suitable person to fix the bug was still to be found.

[2]https://bugs.eclipse.org/bugs/show_bug.cgi?id=221068

Bug 221068 - [xslt][launcher] XSL Parameter variables that use Eclipse Variables don't

Status: RESOLVED FIXED
Product: WTP Source Editing
Component: wst.xsl
Version: unspecified
Platform: PC All
Importance: P3 normal (vote)
Target Milestone: 3.1
Assigned To: Doug
QA Contact: David Williams
Reported: 2008-03-02 12:37 EST by David Carver
Modified: 2009-04-30 14:25 EDT (History)
CC List: 1 user (show)

David Carver 2008-03-02 12:37:31 EST

```
When setting up a launch configuration, and trying to set an XSLT parameter to
use an eclipse variable, the eclipse variable is passed through without being
expanded first.   This causes the wrong or unexpected value to be sent in.
(i.e. if {workspace_loc} is selected as the eclipse variable.   the text
{workspace_loc} is passed through to the XSLT).
```

Who	When	What	Removed	Added
doug.satchwell	2008-03-10 15:47:26 EDT	CC		doug.satchwell
		Status	NEW	RESOLVED
		Resolution	---	FIXED
		Target Milestone	---	0.5 M6
d_a_carver	2008-03-10 16:12:53 EDT	Status	RESOLVED	REOPENED
		Resolution	FIXED	---
d_a_carver	2008-03-10 16:13:36 EDT	Assignee	wtp.inc-inbox	doug.satchwell
		Status	REOPENED	NEW
d_a_carver	2008-03-10 16:13:46 EDT	Status	NEW	ASSIGNED
d_a_carver	2008-03-10 16:14:14 EDT	Status	ASSIGNED	RESOLVED
		Resolution	---	FIXED
d_a_carver	2008-06-05 21:40:41 EDT	Component	incubator	wtp.inc.xsl
webmaster	2009-03-31 10:56:33 EDT	Component	wtp.inc.xsl	wst.xsl
		Product	WTP Incubator	WTP Source Editing
		Target Milestone	0.5 M6	---
d_a_carver	2009-04-30 14:25:06 EDT	Target Milestone	---	3.1

Fig. 1. Reassigned and refined Bug Report of Eclipse Project with BugID 221068.

B. Multi-label Learning

Multi-label learning refers to the task of assigning one or more labels to a data instance. Traditional classification only assigns one label to an instance. However, in many situations, one instance could have more than one label. In our bug report field reassignment and refinementprediction problem, one bug report could have several of its fields reassigned and refined. For example, in Figure 1, the bug report has 4 types of field reassignments and refinements, i.e., product, component, fixer, and status field reassignments and refinements.

Formally, multi-label learning is defined as follows. Let χ denotes the input space (i.e., bug report collection) and let L denote the set of labels (i.e., 8 types of bug report field reassignment and refinement). Given a multi-label training dataset $D = \{(X_i, Y_i)\}_{i=1}^{n}$, where $X_i \in \chi$ denotes a bug report, and $Y_{X_i} = \{0, 1\}^{|L|}$ ($Y_{X_i}(l) = 1$ indicates that the bug report X_i is assigned to the lth label (i.e., one of the field reassignment and refinement types) and $Y_{X_i}(l) = 0$ indicates otherwise), the goal of multi-label classification is to build a model $h : \chi \rightarrow 2^{|L|}$, which is used to predict the proper label set for a new instance.

ML.KNN [13] is one of the state-of-the-art algorithms in the multi-label learning literature. To infer the label set for a new instance (i.e., bug report) X_{new}, ML.KNN follows three steps: the computation of membership counting scores, the computation of ML.KNN scores, and the assignment of labels. We describe each of these steps in the following subsections.

1) Membership Counting Score

ML.KNN first identifies the k-nearest neighbors $knn(X_{new})$ of the new instance X_{new} from the training dataset. For each label l in the label set L, we count the number of instances assigned to label l in $knn(X_{new})$. Formally, we denote membership counting score $C_{X_{new}}(l)$ as the number of instances assigned to label l, i.e.,

$$C_{X_{new}}(l) = \sum_{b' \in knn(X_{new})} Y_{b'}(l), l \in L \tag{1}$$

2) ML.KNN Score

With the membership counting score $C_{X_{new}}(l)$ for each label l, we consider two events: H_1^l is the event that X_{new} is assigned to l, and H_0^l is the event X_{new} is not assigned to l. Moreover, E_m^l denotes the event that there are exactly m instances that are assigned to label l, among the k nearest neighbors of X_{new}. Then, the ML.KNN score for l is the probability that the event X_{new} is assigned to l, given that exactly $C_{X_{new}}(l)$ instances are assigned to label l, i.e.,

$$ML.KNN_{X_{new}}(l) = P(H_1^l | E_{C_{X_{new}}(l)}^l) \tag{2}$$

From Equation (2), and using Bayes rule, we can derive:

$$ML.KNN_{X_{new}}(l) = \frac{P(H_1^l) \times P(E_{C_{X_{new}}(l)}^l | H_1^l)}{\sum_{i \in \{0,1\}} P(H_i^l) \times P(E_{C_{X_{new}}(l)}^l | H_i^l)} \tag{3}$$

The parameters of $P(H_1^l)$, $P(H_1^0)$, $P(E_m^l | H_1^l)$, and $P(E_m^0 | H_1^0)$ can be inferred from the training dataset. The details of the inference process is available in [13].

3) Label Assignment

After the ML.KNN score for each label l is obtained, to decide whether a label should be assigned to X_{new}, ML.KNN uses the following heuristics: if $P(H_1^l) \times P(E_{C_{X_{new}}(l)}^l | H_1^l) > P(H_1^0) \times P(E_{C_{X_{new}}(l)}^l | H_1^0)$, then l is assigned to X_{new}.

III. Overall Framework

Figure 2 shows the overall framework of *Im-ML.KNN*. The framework includes two phases: the model building phase and the prediction phase. In the model building phase, our goal is to build a composite model *MLComposer*, from historical bug reports, which have known bug report field reassignment and refinement information. In the prediction phase, this classifier is used to predict the fields that will get reassigned and refined for a new bug report.

Our framework first extracts features from the set of training bug reports (i.e., bug reports with known field reassignments and refinements) (Step 1). Features are various quantifiable characteristics of bug reports that could potentially differentiate reports for different fields reassignment. In this paper, we consider 2 types of features: meta features and textual features.[3] Next, we analyze the history of the training bug reports, and extract the field reassignment and refinement information (Step 2). Each field corresponds to a label, and in total we have 8 labels which corresponds to 8 types of bug report field reassignments and refinements (i.e., product, component, severity, priority, OS, version, fixer, and status reassignment).[4] The training set is constructed after the feature and label extraction.

Next, our framework constructs 3 multi-label classifiers based on labels and different features of the training set (Step

[3] For more details, please refer to Section IV-A.

[4] For more details, please refer to Section IV-B.

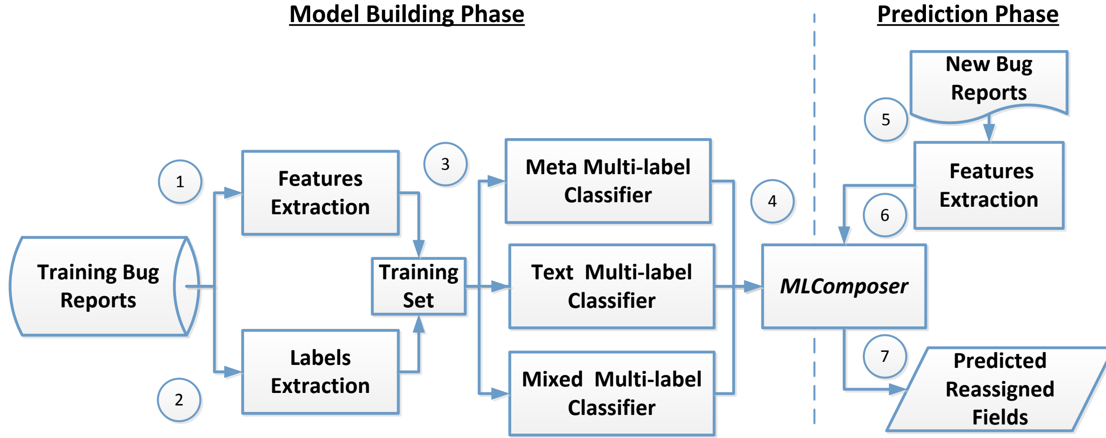

Fig. 2. Overall Framework of *Im-ML.KNN*.

3). In this paper, we use ML.KNN [13] to construct the 3 multi-label classifiers. The meta multi-label classifier is built based on the meta features of bug reports. The text multi-label classifier is built based on the textual features of bug reports. The mixed multi-label classifier is built based on the two types (i.e., meta features and textual features) of features of bug reports. A multi-label classifier is a machine learning model, which assigns a set of labels (in our case: bug report fields that would get reassigned and refined) to a data point (in our case: a bug report) based on its features. We then combine the 3 classifiers together to construct a *MLComposer* classifier (Step 4).

After *MLComposer* is constructed, it is used in the prediction phase to predict the fields that will get reassigned and refined in a new bug report. For each such bug report, we first extract features from it as we do in the model building phase (Step 4). Then, we input the features to *MLComposer* (Step 5). This step outputs the prediction results, which is a set of labels corresponding to the bug report fields that get reassigned and refined.

IV. Feature & Label Extraction

In this section, we first describe the features we extracted from bug reports in Section IV-A. Next, we present the way we extract the labels from the training bug reports in Section IV-B.

A. Feature Extraction

A bug report contains a large amount of useful information, such as its textual description, and the values of its many fields. To predict which bug fields will be reassigned and refined, we extract many features from bug reports. We divide them into 2 categories: meta features and textual features.

1) Meta Features

Meta features refer to the non-textual fields of a bug report, e.g., reporter, assignee, product, component, etc. These fields are important for bug triaging and fixing [2], [3], [16]. Table I presents the meta features which are used to predict which bug fields get reassigned and refined. We extract 9 fields, and we record the values of these fields when a bug is reported - before any reassignments and refinements (if any). Notice that

TABLE I
Meta Features for Bug Report Field Reassignment.

Features	Description	Example
Reporter	The developer who reports the bug.	d_a_carver
Assignee	The assignee before this field is reassigned and refined.	wtp.inc-inbox
Product	The product before this field is re-assigned and refined.	WTP Incuba-tor
Component	The component before this field is reassigned and refined.	incubator
Severity	The severity before this field is re-assigned and refined.	normal
Priority	The priority before this field is re-assigned and refined.	p3
OS	The OS before this field is reas-signed and refined.	All
Version	The product before this field is re-assigned and refined.	unspecified
Platform	The platform before this field is reassigned and refined.	PC

the value of the reporter field is unchanged for the whole life cycle of a bug report- there is no reassignment and refinement for this field. In Table I, column **Example** corresponds to the values of the fields of the example bug report shown in Figure 1. This bug report has product, component, fixer, and status reassignments and refinements, so we trace the mixed values of these 4 fields from its bug history. For the other fields, we use the values recorded in the final bug report.

2) Textual Features

We extract the text in the summary and description fields, and then we tokenize them, remove the stop words, stem them (i.e., reduce them to their root form, e.g., write and written are reduced to writ) using the Porter stemmer, and represent them as TD-IDF (i.e., term frequency.inverse document frequency) vectors [17]. Formally, we represent terms in the ith bug report as a vector of term weights denoted by $b_i = \langle w_{i,1}, w_{i,2} ... w_{i,v} \rangle$. The weight $w_{i,j}$ denotes the TD-IDF score [17] for the jth term in the ith bug report, which is computed as follows:

$$w_{i,j} = \frac{tf_{i,j}}{Num\ of\ Terms\ in\ b_i} \times log(\frac{Num\ of\ BugReports}{df_j}) \quad (4)$$

In the above equation, b_i denotes the ith bug report in the bug report collection, $tf_{i,j}$ denotes the term frequency of the jth term in the ith bug report, df_i denotes the document frequency of the jth term. Term frequency $tf_{i,j}$ refers to the number of times the jth term appears in the ith bug report. The document frequency of the jth term refers to the number of bug reports in which the jth term appears in. To reduce noise, we remove terms which appear less than 10 times in the bug report collection.

B. Label Extraction

For the training bug reports, we need to extract the field reassignment and refinement information from the bug report history. Each field corresponds to a label, and in this paper, we consider 8 types of bug report field reassignments and refinements considered in our previous study [9], i.e., component, product, severity, priority, OS, version, fixer, and status reassignment and refinement. For each of the training bug reports, we parse its history, and check whether any of its 8 fields got reassigned and refined. If a field was reassigned and refined, we set the corresponding label of the field to be "1"; otherwise "0". For example, in Figure 1, we notice its product, component, fixer, and status fields get reassigned and refined. Thus, we set the corresponding labels of product, component, fixer, and status to be "1", and the remaining labels to be "0".

V. MLComposer: A Composite Method

ML.KNN is used to construct a multi-label classifier to predict the fields which would get reassigned and refined for bug reports. However, as we noted, for each type of field reassignment and refinement (except for the fixer reassignment), the number of bug reports whose fields have been reassigned and refined is much smaller than the number of bug reports without reassignment, i.e., the class imbalance phenomenon [14] is observed. For example, in Eclipse, only 9.76%, 18.44%, 9.19%, and 8.14% of bug reports have their product, component, severity and status fields reassigned and refined; in Mozilla, only 19.27%, 24.68%, 7.23%, and 11.39% of bug reports have their product, component, severity and status fields reassigned and refined [9]. To adapt ML.KNN to work well in imbalanced multi-label data (i.e., having much less bug reports with reassigned and refined fields than bugs without reassigned and refined fields), and also considering that we have multiple types of features, i.e., meta features and textual features, in this section, we propose MLComposer which combines 3 multi-label classifiers (ML.KNN classifiers) built on different types of features and considers the imbalance phenomenon. In this section, we first define three sets of scores outputted by the three classifiers. Next, we describe how we combine these scores together to construct the *MLComposer* classifier.

A. Feature Scores

As illustrated in Figure 2, our proposed framework has 3 different multi-label classifiers (i.e., ML.KNN classifiers built using each of the three feature types). Let us refer to them as $ML.KNN_{Meta}$, $ML.KNN_{Text}$, and $ML.KNN_{Mixed}$. Given an unknown bug report, $ML.KNN_{Meta}$, $ML.KNN_{Text}$, and $ML.KNN_{Mixed}$ output the following meta scores, text scores, and mixed scores, respectively:

Definition 1: (**Meta Scores.**) Consider a set of training bug reports BR, its corresponding set of meta feature values $Meta$, and its corresponding set of labels L. We build a ML.KNN classifier $ML.KNN_{Meta}$ trained on $Meta$. For a new bug report br, for each label $l \in L$, we use $ML.KNN_{Meta}$ to get the likelihood that br will be assigned to the label l (i.e., the field corresponds to label l would get reassigned and refined). We refer to the likelihood score as the **meta score** for label l, and denote it as $ML.KNN_{Meta}(br, l)$.

Definition 2: (**Text Scores.**) Consider a set of training bug reports BR, its corresponding set of text feature values $Text$, and its corresponding set of labels L. We build a ML.KNN classifier $ML.KNN_{Text}$ trained on $Text$. For a new bug report br, for each label $l \in L$, we use $ML.KNN_{Text}$ to get the likelihood that br will be assigned to the label l. We refer to the likelihood score as the **text score** for label l, and denote it as $ML.KNN_{Text}(br, l)$.

Definition 3: (**Mixed Scores.**) Consider a set of training bug reports BR, its corresponding set of meta and text feature values $Mixed$, and its corresponding set of labels L. We build a ML.KNN classifier $ML.KNN_{Mixed}$ trained on $Mixed$. For a new bug report br, for each label $l \in L$, we use $ML.KNN_{Mixed}$ to get the likelihood that br will be assigned to the label l. We refer to the likelihood score as the **mixed score** for label l, and denote it as $ML.KNN_{Text}(br, l)$.

B. MLComposer

In this section, we propose *MLComposer*, a composite method which uses all of these 3 scores, and also considers the imbalance phenomenon in bug report field reassignments and refinements. A linear combination of meta scores, text scores, and mixed scores is used to compute the final *MLComposer* scores.

Definition 4: (***MLComposer* Score.**) Consider a training bug report collection BR, and its corresponding multi-label classifiers for meta, description, and mixed features ($ML.KNN_{Meta}$, $ML.KNN_{Text}$, and $ML.KNN_{Mixed}$), respectively. For a new bug report br, for each label $l \in L$ we compute its corresponding meta, text and mixed scores, then its MLComposer score, denoted as $MLComposer(br, l)$, which is a linear combination of the 3 scores, is defined as follows:

$$\begin{aligned} MLComposer(br, l) \quad = \quad & \alpha \times ML.KNN_{meta}(br, l) + \\ & \beta \times ML.KNN_{text}(br, l) + \\ & \gamma \times ML.KNN_{mixed}(br, l) \quad (5) \end{aligned}$$

In the above equations, $\alpha \in [0, 1]$, $\beta \in [0, 1]$, and $\gamma \in [0, 1]$, and $\alpha + \beta + \gamma = 1$.

To deal with imbalanced data, *MLComposer* introduces a threshold for every label. Each threshold is independently fine tuned based on a sample of a training data so that the bias introduced by the imbalanced data can be offset. For each label

$l \in L$, we define a threshold $threshold_l$. To decide whether a label l is assigned to a new bug report (aka. an instance) br, we follow the following equation:

$$Label_{br}(l) = \begin{cases} 1, & if\ MLComposer(br, l) \geq threshold_l \\ 0, & Otherwise \end{cases} \quad (6)$$

The value of $threshold_l$ for each label l can be trained from the training bug report collection. To automatically produce good α, β, γ, and $threshold_l$ ($l \in L$) values for *MLComposer*, we propose a greedy algorithm.

Figure 1 presents the detailed steps to estimate good α, β, γ, and $threshold_l$ values. We input a bug report collection BR, label set L (i.e., various types of bug report field reassignment), sample size $Sample$, meta features $Meta$, textual features $Text$, mixed features $Mixed$, and the number of neighbors K. We first sample a small bug report collection $Samp_{BR}$ according to the sample size $Sample$ (Line 12). Next, we initialize α, β, γ, and $threshold_l$ ($l \in L$) values to 0 at Line 13. Then, we build the classifiers (i.e., $ML.KNN_{Meta}$, $ML.KNN_{Text}$, and $ML.KNN_{Mixed}$) for meta features, textual features, and mixed features using BR, and compute their corresponding meta, text, and mixed scores of bug reports in $Samp_{BR}$ at Lines 14, 15 and 16, respectively. Next, we incrementally increase α, β, γ, and $threshold_l$ values (Lines 17 to 21). We increase α from 0 to 1, and $beta$ from 0 to $(1-\alpha)$, in 0.1 increments. The value of γ is set as $(1-\alpha-\beta)$. We increase the $threshold_l$ for each label l in L from 0 to 1, in 0.01 increments. We use a finer granularity step to tune $threshold_l$ since it directly decides whether a bug report will be assigned to label l. We use a coarser granularity step to tune α, β, and γ values to reduce the computational cost in the tuning process. For each configuration of α, β, γ, and $threshold_l$ values, we build a composite model and compute the resultant F-measure score using bug reports in $Samp_{BR}$ (Lines 20 to 26). Finally, Algorithm 1 returns α, β, γ, and $threshold_l$ values resulting in the best average F-measure scores across all the labels l in L (Line 32).

Algorithm 1 Estimation of Good α, β, γ, and $threshold_l$ ($l \in L$) Values in *MLComposer*.

1: **Estimatevalue(**BR**,** L**,** $Sample$**,** $Meta$**,** $Text$**,** $Mixed$**,**K**)**
2: **Input:**
3: BR: Training bug report collection and their labels
4: L: Label set
5: $Sample$: Sample size (default value: 10% of $|BR|$)
6: $Meta$: Meta features
7: $Text$: Textual features
8: $Mixed$: Mixed features
9: K: Number of neighbors for ML.KNN
10: **Output:** α, β, γ, and $threshold_l$ ($l \in L$)
11: **Method:**
12: Sample a bug report collection $Samp_{BR}$ of size $Sample$ from BR;
13: $\alpha = 0, \beta = 0, \gamma = 0$, $threshold_l = 0$ ($l \in L$);
14: Build $ML.KNN_{Meta}$ for $Meta$ and BR, compute meta scores for each bug report in $Samp_{BR}$;
15: Build $ML.KNN_{Text}$ for $Text$ and BR, compute text scores for each bug report in $Samp_{BR}$;
16: Build $ML.KNN_{Mixed}$ for $Mixed$ and BR, compute mixed scores for each bug report in $Samp_{BR}$;
17: **for all** α from 0 to 1, every time increase α by 0.1 **do**
18:   **for all** β from 0 to $(1-\alpha)$, every time increase β by 0.1 **do**
19:     $\gamma = 1 - \alpha - \beta$;
20:     **for all** labels $l \in L$ **do**
21:       **for all** $threshold_l$ from 0 to 1, every time increase $threshold$ by 0.01 **do**
22:         **for all** Bug report br in $Samp_{BR}$ **do**
23:           Compute *MLComposer* score according to Definition 4;
24:           Decide whether br is assigned to label l using Equation 6;
25:         **end for**
26:         Compute the F-measure score of the lth label;
27:       **end for**
28:     **end for**
29:     Compute the average F-measure score across all the labels in L.
30:   **end for**
31: **end for**
32: **Return** α, β, γ, and $threshold_l$ ($l \in L$), which maximize average F-measure score across all the labels in L.

VI. Experiments and Results

In this section, we evaluate *Im-ML.KNN*. The experimental environment is a Windows 7, 64-bit, Intel(R) Xeon(R) 2.53GHz server with 24GB RAM. We first present our experiment setup and 6 research questions. We then present our experiment results that answer each of the 6 research questions.

A. Experiment Setup

Table II shows the statistics of the 4 projects we use to evaluate the performance *Im-ML.KNN* which are also used in our previous empirical study [9]. All of the bug reports and data are downloaded from their corresponding bug tracking systems. We collect all bug reports with the status "resolved", "closed", and "fixed" following previous studies [5]–[7], [12], [18]. In Table II, columns **Time** and **# Report** correspond to the time periods the collected bug reports are reported and the number of collected reports, respectively. In total, we collect 190,558 bug reports. Columns **# reporter**, **# fixer**, **# product**, **# component**, **# version**, **# OS**, and **# platform** correspond to the number of unique values of the different fields. Notice that the values of these fields are recorded at the time the bug report is submitted, i.e., the values of all of these fields are recorded before the fields of the bug report are reassigned and refined.

We use 10 times ten-fold cross validation [19] to evaluate the performance of *Im-ML.KNN*. We randomly divide the dataset into 10 folds. Of these 10 folds, 9 folds are used to train the model, while the remaining one fold is used to evaluate the performance. The whole process iterates 10 times. The overall performance score across the 10 iterations is reported. Also, we run ten-fold cross validation 10 times, and record the average performance to further reduce the bias due to training set selection. Cross validation is a standard evaluation setting, which is widely used in software engineering studies, c.f., [5], [20]–[24].

TABLE II
STATISTICS OF COLLECTED BUG REPORTS.

Project	Time	# Reports	# Reporter	# Fixer	# Product	# Component	# Version	# OS	# Platform
OpenOffice	2002-05-17 – 2013-04-07	42,169	5,451	701	140	106	546	45	12
Netbeans	2008-01-01 – 2013-03-13	46,345	5,709	323	112	684	43	26	7
Eclipse	2008-01-01 – 2011-07-19	50,639	5,824	1,021	143	702	220	31	6
Mozilla	2009-06-23 – 2012-02-23	51,405	3,536	696	51	620	107	36	10

We implement *Im-ML.KNN* on top of Mulan [25], a multi-label learning Java toolkit. By default, we set the number of neighbors $K = 10$ as [13], and the sample size to 10% of the number of bug reports in the training dataset. For ML.KNN, we directly use its implementation in Mulan, and we set $K = 10$ which is the same as *Im-ML.KNN*.

Lamkanfi et al. propose the usage of Naive Bayes to predict whether a component field would be reassigned and refined [6]. The output of their method is only reassigned and refined or non-reassigned and refined for the component field of a bug report, which corresponds to the single-label learning problem in machine learning literature [26]–[28]. Different from their work, our works focus on predicting the sets of bug report fields which would get reassigned and refined, the output of our method is multiple labels which represents the fields of bug reports, which is a typical multi-label learning problem. To adapt Lamkanfi et al.'s method, we use their method to predict the reassignment and refinement of each field independently, and repeat the process 8 times. In this way, we build 8 classifiers using Naive Bayes, and each classifier predict one type of field reassignment. For example, classifier 1 could correspond to the classifier which predict whether the component field of a bug report would get reassigned and refined, classifier 2 could correspond to the classifier which predict whether the product field of a bug report would get reassigned and refined.

In multi-label learning literature, Tsoumakas et al. propose HOMER algorithm which also considers the class imbalance problem [15]. HOMER builds a hierarchy of multi-label classifiers by leveraging a balanced clustering algorithm, each one dealing with a much smaller set of labels compared to the total L labels, and a more balanced example distribution. Tsoumakas et al. use Naive Bayes as the underlying classifier of HOMER (referred to in this paper as HOMER-NB). In this paper, we also compare Im-ML.KNN with HOMER-NB.

B. Evaluation Metrics

To measure the performance of *Im-ML.KNN*, we use precision, recall, and F-measure scores as our evaluation metrics. For each type of bug report field reassignment, and we refer to this type of bug report field reassignment and refinementas a label l in multi-label learning literature. Give a label l in the label set L, for an instance (aka. a bug report), there are four outcomes: An instance is assigned to the label l when it is truly assigned to l (true positive, TP_l); it assigned to label l when it is not actually assigned to l (false positive, FP_l); it is not assigned to label l when it is actually assigned to l (false negative, FN_l); or it is not assigned to label l when it actually is not assigned to l (true negative, TN_l). Based on these possible outcomes, we compute its own F-measure, precision, and recall, i.e., we have 8 F-measures, precisions, and recalls, one for each label l:

- **Precision for l:** the proportion of bug reports (instances) that are correctly labeled as l among those labeled as l:

$$P_l = \frac{TP_l}{TP_l + FP_l} \tag{7}$$

- **Recall for l:** the proportion of bug reports labeled as l that are correctly labeled.

$$R_l = \frac{TP_l}{TP_l + FN_l} \tag{8}$$

- **F-measure for l:** a summary measure that combines both precision and recall for label l - it evaluates whether an increase in precision (recall) outweighs a reduction in recall (precision).

$$F_l = \frac{2 \times P_l \times R_l}{P_l + R_l} \tag{9}$$

In addition to the 8 precision, recall, and F-measure scores, we also compute in the average precision, recall, and F-measure scores over the 8 precision, recall, and F-measure scores.

$$\begin{aligned} Ave.P &= \tfrac{1}{|L|}\sum_{l \in L} P_l \\ Ave.R &= \tfrac{1}{|L|}\sum_{l \in L} R_l \\ Ave.F &= \tfrac{1}{|L|}\sum_{l \in L} F_l \end{aligned} \tag{10}$$

Notice that the average precision, recall, and F-measure measure the prediction performance across all the $|L|$ labels, which are also used in previous software engineering studies [20], [29], and many multi-label learning studies [26]–[28].

C. Research Questions

We would like to answer the following research questions:

RQ1 **What is the F-measure of *Im-ML.KNN*? How much improvement can it achieve over the method proposed by Lamkanfi et al. [6], ML.KNN [13], and HOMER-NB [15]**?

RQ2 **Can the F-measure of Im-ML.KNN outperforms those of its constituent components (i.e., meta classifier, text classifier, and mixed classifier)?**

RQ3 **Do different numbers of neighbors affect the F-measure of *Im-ML.KNN*?**

RQ4 **What are good predictors for bug report field reassignments and refinements? Do the predictors differ for different fields?**

TABLE VI
AVERAGE PRECISION (AVE. PRECISION) AND RECALL (AVE. RECALL) OF IM-ML.KNN, LAMKANFI EL AL.'S METHOD, ML.KNN, AND HOMER-NB.

Project	Algorithms	Ave. Precision	Ave. Recall
OpenOffice	Im-ML.KNN	0.6090	0.6406
	Lamkanfi el al.'s	0.2237	0.8379
	ML.KNN	0.8030	0.5491
	HOMER-NB	0.2711	0.4115
Netbeans	Im-ML.KNN	0.6113	0.5914
	Lamkanfi el al.'s	0.2636	0.6010
	ML.KNN	0.7352	0.4697
	HOMER-NB	0.3378	0.2996
Eclipse	Im-ML.KNN	0.5671	0.5613
	Lamkanfi el al.'s	0.1929	0.8154
	ML.KNN	0.6923	0.4318
	HOMER-NB	0.2279	0.3788
Mozilla	Im-ML.KNN	0.5739	0.5899
	Lamkanfi el al.'s	0.2194	0.2194
	ML.KNN	0.7776	0.4775
	HOMER-NB	0.2588	0.4664

RQ5 What is the effect of varying the amount of training data on the effectiveness of *Im-ML.KNN*?
RQ6 How much time does it take for Im-ML.KNN to run?

Answering RQ1 sheds light on the effectiveness of *Im-ML.KNN* to predict bug report fields that get reassigned and refined, compared to existing state-of-the-art solutions [6], [13]. Answering RQ2 highlights the effectiveness of our approach compared to each of its individual classifiers. Answering RQ3 sheds light on the sensitivity of *Im-ML.KNN* when using various settings of its parameter, i.e., the number of neighbors. The answer of RQ4 presents the top features that best indicate bug report field reassignments and refinements, which can be used by software practitioners to determine, early on, which fields are most likely to be reassigned and refined. The answer to RQ5 determines the impact of reducing the amount of training data on the performance of our approach. The answer to RQ6 examines the model building time and prediction time for Im-ML.KNN.

D. RQ1: F-measure Scores of Im-ML.KNN

Tables III, IV and V present the experimental results of Im-ML.KNN compared with the method proposed by Lamkanfi *et al.* [6], ML.KNN [13], and HOMER-NB [15], respectively. We also list the average precision and recall of Im-ML.KNN, Lamkanfi el al.'s method, ML.KNN and HOMER-NB in Table VI. Notice that for Netbeans, no bug report has its severity reassigned and refined, therefore the precision, recall, and F-measure for severity reassignment and refinement is 0.

Precision and recall are both important metrics for reassigned and refined bug prediction since they measure quality in two aspects. If the precision is low, then the developer would not use the technique, due to a high number of false alarms. On the other hand, if the recall is low, which means that most reassigned and refined bug reports are not successfully detected, developers would also not use the technique. There is a trade off between precision and recall [19]. One can increase precision by sacrificing recall (and vice versa). One simple way to increase recall is to predict all the bug reports as reassigned and refined, then the recall would be 1 but the precision would be 0. In our method, we can sacrifice precision (recall) to increase recall (precision), by manually lowering (increasing) the value of the $threshold_l$ parameter in Equation 6. F-measure, which is a weighted harmonic mean of precision and recall, is often used to judge whether an increase in precision outweighs a loss in recall (and vice versa) [19]. Thus, in many existing papers, e.g., [22], [30]–[32], it is often used as a summary measure. In Figure 1, the $threshold_l$ parameter is automatically tuned to maximize the F-measure for each type of bug report field reassignment and refinementin the training data.

From Table III, we note that the improvement of our method over Lamkanfi *et al.*'s method is substantial. We improve the average F-measure of the method proposed by Lamkanfi *et al.* by 127.17%, 98.30%, 139.39%, and 113.87% for OpenOffice, Netbeans, Eclipse, and Mozilla, respectively. Averaging across the four datasets, the average improvement achieved by *Im-ML.KNN* is 119.69%.

From Table IV, the improvement of our method over ML.KNN is substantial. We improve the average F-measure of ML.KNN by 0.98%, 15.03%, 9.85%, and 10.57% for OpenOffice, Netbeans, Eclipse, and Mozilla, respectively. Averaging across the four datasets, the average improvement achieved by *Im-ML.KNN* is 9.11%. We also notice that ML.KNN does not work well for imbalanced data, for example, ML.KNN's F-measures for predicting status reassignment and refinementare quite low, i.e., 0.4739, 0.0004, 0.0000, and 0.0340 for OpenOffice, Netbeans, Eclipse, and Mozilla, respectively. Our method overcomes the weakness of ML.KNN and it can achieve better results for each type of bug report field reassignment.

From Table V, we note that the improvement of our method over HOMER-NB is substantial. We improve the average F-measure of HOMER-NB by 164.90%, 148.67%, 188.95%, and 141.80% for OpenOffice, Netbeans, Eclipse, and Mozilla, respectively. Averaging across the four datasets, the average improvement achieved by *Im-ML.KNN* is 161.08%. We also notice that HOMER-NB does not work well for OS reassignment, the F-measures for OS reassignment and refinementare quite low, i.e., 0.1242, 0.1171, 0.1040, and 0.2252 for for OpenOffice, Netbeans, Eclipse, and Mozilla, respectively. Notice the numbers of bug reports whose OS get reassigned and refined are quite small compared to other types of field reassignment and refinement– OS reassignment and refinementonly happens for 5.78%, 4.79%, 4.55%, and 12.34% of OpenOffice, Netbeans, Eclipse, and Mozilla bug reports respectively.

E. RQ2: Benefits of Composition

Table VII presents the average F-measure scores of Im-ML.KNN compared to the meta classifier, text classifier, and mixed classifier. We notice the improvement of Im-ML.KNN over the 3 classifiers are substantial. On average across the 4 projects, Im-ML.KNN improves the average F-measure scores of meta classifier, text classifier, and mixed classifier by 8.91%, 164.31%, and 9.11% respectively. The results show that it is beneficial to combine the 3 classifiers.

TABLE III
EXPERIMENT RESULTS OF *Im-ML.KNN* COMPARED WITH THE METHOD PROPOSED BY LAMKANFI EL AL. [6]. WE CONSIDER MACRO-AVERAGE F-MEASURE (AVE. F-MEA.) AND THE F-MEASURE FOR EACH TYPE OF BUG REPORT FIELD REASSIGNMENT. AVE. = AVERAGE, PRO. = PRODUCT, COM. = COMPONENT, SEV. = SEVERITY, PRI. = PRIORITY, VER. = VERSION, FIX. = FIXER, STAT. = STATUS.

Projects	Algorithms	Ave.F-mea	Comp.F-mea.	Pro.F-mea.	Sev.F-mea.	Pri.F-mea.	OS F-mea.	Ver.F-mea.	Fix.F-mea.	Stat.F-mea.
OpenOffice	Im-ML.KNN	0.6204	0.7284	0.7944	0.7551	0.2954	0.3583	0.6522	0.9056	0.4742
	Lamkanfi el al.'s	0.2731	0.2729	0.3044	0.0190	0.1993	0.1165	0.2122	0.6269	0.4335
	Improvement.	**127.17%**	**166.91%**	**160.98%**	**3874.08%**	**48.22%**	**207.51%**	**207.37%**	**44.46%**	**0.38%**
Netbeans	Im-ML.KNN	0.5963	0.9304	0.8693	0.0000	0.2856	0.5306	0.5821	0.7788	0.1971
	Lamkanfi el al.'s	0.3007	0.5064	0.4952	0.0000	0.2792	0.0936	0.0936	0.4132	0.1918
	Improvement.	**98.30%**	**83.73%**	**75.54%**	**0.00%**	**2.28%**	**466.84%**	**521.93%**	**88.49%**	**2.76%**
Eclipse	Im-ML.KNN	0.5597	0.6365	0.7334	0.2577	0.5413	0.6606	0.6341	0.8667	0.1475
	Lamkanfi el al.'s	0.2338	0.3190	0.1858	0.1758	0.2052	0.0904	0.2311	0.5123	0.1505
	Improvement.	**139.39%**	**99.54%**	**294.74%**	**46.60%**	**163.77%**	**630.80%**	**174.36%**	**69.19%**	**-2.01%**
Mozilla	Im-ML.KNN	0.5796	0.7395	0.8123	0.2392	0.4817	0.7301	0.5843	0.8685	0.1813
	Lamkanfi el al.'s	0.2710	0.4174	0.3380	0.1420	0.2206	0.2404	0.1907	0.4463	0.1724
	Improvement.	**113.87%**	**77.18%**	**140.31%**	**68.48%**	**118.36%**	**203.70%**	**206.40%**	**94.59%**	**5.17%**

TABLE IV
EXPERIMENT RESULTS OF *Im-ML.KNN* COMPARED WITH ML.KNN [13].

Projects	Algorithms	Ave.F-mea	Comp.F-mea.	Pro.F-mea.	Sev.F-mea.	Pri.F-mea.	OS F-mea.	Ver.F-mea.	Fix.F-mea.	Stat.F-mea.
OpenOffice	Im-ML.KNN	0.6204	0.7284	0.7944	0.7551	0.2954	0.3583	0.6522	0.9056	0.4742
	ML.KNN	0.6144	0.8272	0.8652	0.7769	0.1075	0.2314	0.7058	0.9271	0.4739
	Improvement.	**0.98%**	**-11.95%**	**-8.18%**	**-2.80%**	**174.67%**	**54.84%**	**-7.59%**	**-2.31%**	**0.05%**
Netbeans	Im-ML.KNN	0.5963	0.9304	0.8693	0.0000	0.2856	0.5306	0.5821	0.7788	0.1971
	ML.KNN	0.5184	0.9407	0.8866	0.0000	0.0141	0.4729	0.5296	0.7845	0.0004
	Improvement.	**15.03%**	**-1.10%**	**-1.95%**	**0.00%**	**1929.89%**	**12.18%**	**9.93%**	**-0.72%**	**49175.00%**
Eclipse	Im-ML.KNN	0.5597	0.6365	0.7334	0.2577	0.5413	0.6606	0.6341	0.8667	0.1475
	ML.KNN	0.5095	0.6456	0.7523	0.0365	0.5274	0.6296	0.6147	0.8699	0.0000
	Improvement.	**9.85%**	**-1.41%**	**-2.51%**	**606.69%**	**2.64%**	**4.93%**	**3.15%**	**-0.36%**	∞
Mozilla	Im-ML.KNN	0.5796	0.7395	0.8123	0.2392	0.4817	0.7301	0.5843	0.8685	0.1813
	ML.KNN	0.5242	0.7834	0.8554	0.0128	0.3221	0.7139	0.5979	0.8743	0.0340
	Improvement.	**10.57%**	**-5.60%**	**-5.04%**	**1770.54%**	**49.55%**	**2.27%**	**-2.27%**	**-0.67%**	**433.80%**

TABLE V
EXPERIMENT RESULTS OF *Im-ML.KNN* COMPARED WITH HOMER-NB [15].

Projects	Algorithms	Ave.F-mea	Comp.F-mea.	Pro.F-mea.	Sev.F-mea.	Pri.F-mea.	OS F-mea.	Ver.F-mea.	Fix.F-mea.	Stat.F-mea.
OpenOffice	Im-ML.KNN	0.6204	0.7284	0.7944	0.7551	0.2954	0.3583	0.6522	0.9056	0.4742
	HOMER-NB	0.2342	0.3551	0.3130	0.0203	0.1850	0.1242	0.2554	0.2411	0.3803
	Improvement.	**164.90%**	**105.11%**	**153.84%**	**3621.76%**	**59.70%**	**188.58%**	**155.35%**	**275.69%**	**24.69%**
Netbeans	Im-ML.KNN	0.5963	0.9304	0.8693	0.0000	0.2856	0.5306	0.5821	0.7788	0.1971
	HOMER-NB	0.2398	0.5099	0.3289	0.0000	0.1971	0.1171	0.1060	0.2291	0.1907
	Improvement.	**148.67%**	**82.46%**	**164.28%**	**0.00%**	**44.89%**	**353.12%**	**449.02%**	**239.96%**	**3.36%**
Eclipse	Im-ML.KNN	0.5597	0.6365	0.7334	0.2577	0.5413	0.6606	0.6341	0.8667	0.1475
	HOMER-NB	0.1937	0.2891	0.2088	0.1557	0.2491	0.1040	0.1937	0.2031	0.1468
	Improvement.	**188.95%**	**120.16%**	**251.25%**	**65.55%**	**117.33%**	**535.32%**	**227.35%**	**326.75%**	**0.48%**
Mozilla	Im-ML.KNN	0.5796	0.7395	0.8123	0.2392	0.4817	0.7301	0.5843	0.8685	0.1813
	HOMER-NB	0.2397	0.4050	0.3972	0.1446	0.2441	0.2252	0.2460	0.0950	0.1613
	Improvement.	**141.80%**	**82.58%**	**104.53%**	**65.48%**	**97.37%**	**224.19%**	**137.55%**	**814.01%**	**12.40%**

TABLE VII
AVERAGE F-MEASURE OF IM-ML.KNN COMPARED WITH EACH OF 3 MULTI-LABEL CLASSIFIERS, I.E., META CLASSIFIER, TEXT CLASSIFIER, AND MIXED CLASSIFIER.

Project	Im-ML.KNN	Data Type	ML.KNN	Improvement
OpenOffice	0.6204	Meta	0.6197	**0.11%**
		Text	0.1972	**214.60%**
		Mixed	0.6144	**0.98%**
Netbeans	0.5963	Meta	0.5333	**11.81%**
		Text	0.2691	**121.59%**
		Mixed	0.5184	**15.03%**
Eclipse	0.5597	Meta	0.5061	**10.59%**
		Text	0.1977	**183.11%**
		Mixed	0.5095	**9.85%**
Mozilla	0.5796	Meta	0.5124	**13.11%**
		Text	0.2436	**137.93%**
		Mixed	0.5242	**10.57%**

To investigate which classifier plays an important role in Im-ML.KNN, we presents the average weights of the meta classifier, text classifier, and mixed classifier in Table VIII. Note that we run 10-fold cross-validation 10 times, and for each fold we have a set of weights. In total, we have 100 sets of weights, and we record the average weights across the 100 sets. From Table VIII, we observe that the mixed classifier plays the most important role in Im-ML.KNN, followed by meta classifier and text classifier.

F. RQ3: Sensitivity of Im-ML.KNN on Optimal Setting

Since the number of neighbors can impact the performance of the algorithm, we investigate the effect of varying the number of neighbors K on the performance of *Im-ML.KNN*.

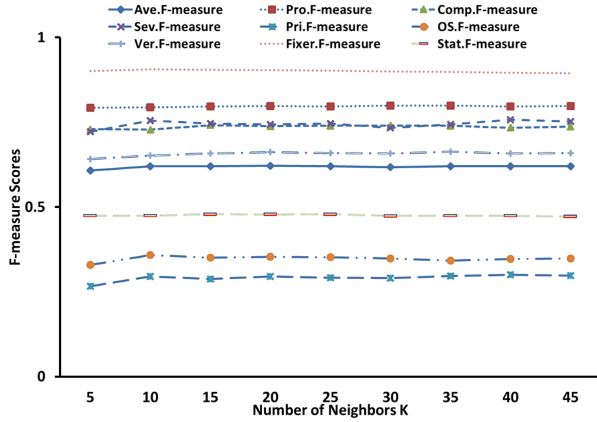

Fig. 3. Experiment Results of Im-ML.KNN for OpenOffice with Number of Neighbors K Varied from 5 to 45. Ave. = Average, Pro. = Product, Comp. = Component, Sev. = Severity, Pri. = Priority., Ver. = Version, Stat. = Status.

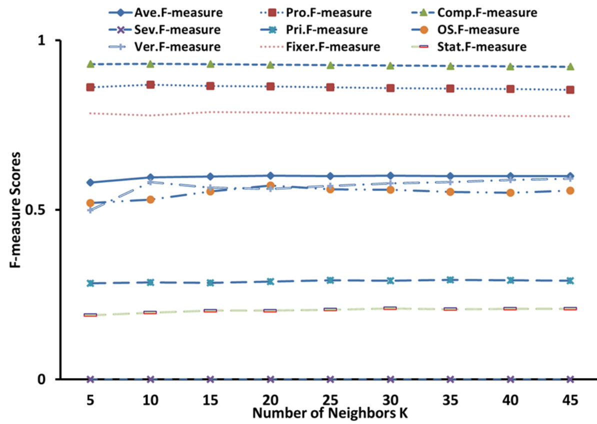

Fig. 4. Experiment Results of Im-ML.KNN for Netbeans with Number of Neighbors K Varied from 5 to 45.

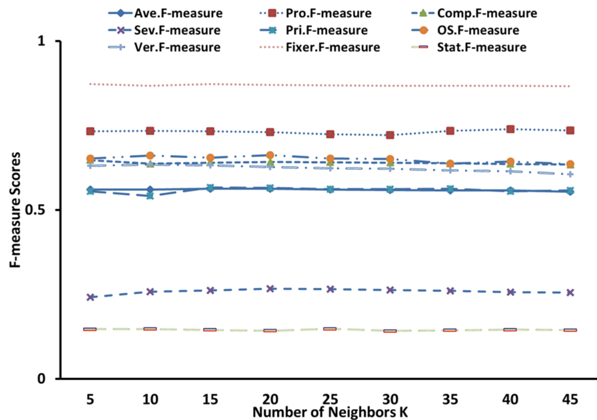

Fig. 5. Experiment Results of Im-ML.KNN for Eclipse with Number of Neighbors K Varied from 5 to 45.

TABLE VIII
AVERAGE WEIGHTS FOR THE META CLASSIFIER, TEXT CLASSIFIER, AND MIXED CLASSIFIER.

Projects	Meta Classifier	Text Classifier	Mixed Classifier
OpenOffice	0.35	0.14	0.51
Netbeans	0.24	0.16	0.60
Eclipse	0.28	0.18	0.54
Mozilla	0.30	0.23	0.47

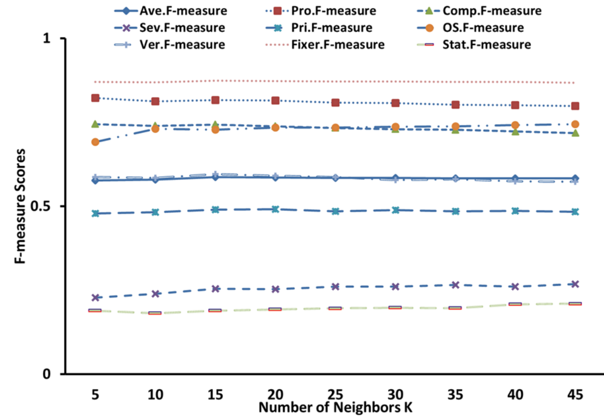

Fig. 6. Experiment Results of Im-ML.KNN for Mozilla with Number of Neighbors K Varied from 5 to 45.

We vary the number of neighbors (i.e., K in Algorithm 1) from 5 to 45.

We plot the resultant F-measure scores for OpenOffice, Netbeans, Eclipse, and Mozilla in Figure 3, 4, 5, and 6, respectively. For Openoffice, the average F-measure scores vary from 0.6073 ($K = 5$) to 0.6215 ($K = 20$). For Netbeans, the average F-measure scores vary from 0.5814 ($K = 5$) to 0.0.6007 ($K = 20$). For Eclipse, the average F-measure scores vary from 0.5540 ($K = 45$) to 0.5629 ($K = 15$). For Mozilla, the average F-measure scores vary from 0.5763 ($K = 5$) to 0.5860 ($K = 15$). The results show that the performance of *Im-ML.KNN* is relatively stable with various numbers of neighbors. Across the 4 projects, *Im-ML.KNN* achieves the best performance with the number of neighbors K set to 15 or 20.

G. *RQ4: Indicators of Bug Field Reassignment*

From the bug reports, we extract thousands of features (i.e., meta features and textual features). To understand which features are important to classify field reassignment, we extract discriminative features from the thousands of features. We extract top-5 features per label based on their information gain scores [19].

Table IX presents the top-10 most discriminative features. We notice that the meta features (e.g., product, component, assignee) make up most of the top-10 features. Notice that we use the value of the meta features before they are reassigned and refined (i.e., the first values of these features). We find that the first/initial values of these features can be good indictors to predict which fields would get reassigned and refined. Among the 4 projects, *product*, *component*, *reporter*, and *assignee* are the 4 most important meta features related to various types

TABLE IX
TOP-10 DISCRIMINATIVE FEATURES BASED ON INFORMATION GAIN SCORES. THE META FEATURES ARE UNDERLINED.

Projects	Product	Component	Severity	Priority	OS	Version	Fixer	Status
OpenOffice	Product Reporter Assigee Component Version chart ac star checkap diction	Product Reporter Assigee Component Version chart ac checkap star OS	Severiry Version Reporter Assignee OS Component Product attachm Platorm gbuild	Reporter Assignee Priority Version Product Component OS crash Platorm docum	Reporter OS Assignee Version Product Component Platform docum writer) spreadsheet	Product Reporter Assignee Version Component chart ac star checkap sockes	Assignee Reporter Product Version Component OS docum Platform http distribut	Product Reporter Assignee Component Version chart ac star checkap diction
Netbeans	Product Component Reporter Assignee Severity Version OS bpel Platform client	Component Severity Reporter Product Assignee Version Plaform OS attachm sever	null null null null null null null null null null	Reporter Component Assignee Priority Product Version project OS sunflow start	Reporter OS Component Assignee Product Version stacktrac Severity environm comment	Version Reporter Component Assignee Product disgram macksic Platform Priority layoutdesign	Assignee Reporter Component Product Version Severity diagram OS scenari makeproj	Product Component Reporter Assignee Severity Version OS bpel Platform client
Eclipse	Assignee Component Reporter Product Version consmo jsdt cmdb wikitext inform	Component Assignee Reporter Product Version cosmo step reproduc build inform	Reporter Assignee Component Product Version Severity mylyn task OS tptp	Reporter Assignee Component Product Version clon init creat tptp OS	Reporter Assignee Component Platform Product OS Version descript coco birt	Reporter Assignee Component Product Version identif cosmo reproduc mozil geck	Assignee Reporter Component Product Version OS step descript build reproduc	Reporter Assignee Component Product Version OS respons reassign clon atricl
Mozilla	Component Reporter Product Assignee "pag" local attach thank text fil	Component Reporter Product Assignee "firefox" Version thank html local text	Reporter Component Assignee Product OS Version Severity agent reproduc identif	Component Reporter Product Assignee Version OS thank commis pleas min	Reporter OS Component Assignee Product geck identif agent Platform Version	Reporter Component Product "agent" "identif" Version geck Assignee reproduc user	Assignee Reporter Component Product Version OS transl patch repositor attach	Reporter Component Assignee Product Version mim altern localecod plain iphon

of field reassignment. For example, to predict whether the product field would get reassigned and refined, the value of the meta feature *product* is a good indictor, since the initial value of the *product* field maybe wrong, corresponds to a non-existent product, or is a default value that most likely would get changed later. Aside from these meta features, some textual features, corresponding to stemmed words that appear in bug reports, are also good indicators to various field reassignments and refinements. Note that, in Table IX, the set of top-10 features for Netbeans corresponding to label Severity is empty; this is the case since none of the Netbeans bug reports has its severity field reassigned and refined.

To further investigate why the most discriminative features differs for different types of bug report field reassignment and refinement, we also perform a simple qualitative analysis. Notice that the fields in a bug report are related; some feature combinations are good indicators for the field reassignment and refinement. For example, the combinations of product, component are important indicators to find the suitable fixers during the bug triaging process [6]. For some product and component combinations, it is easy to find suitable fixers. For some other combinations, it might be hard to find suitable fixers, which results in bugs being "tossed" among multiple fixers. Thus, the combinations of the product and component fields can help to decide whether the fixer field would get reassigned. For example, in the collected dataset of OpenOffice, the combination of product ="Writer", and component ="code" appears 2,308 times, and there are total of 1,892 times that the fixer fields are reassigned under this combination. In the collected dataset of Mozilla, the combination of product ="mozilla.org" and component ="Server Operations" appears 3,066 times, and there are a total of 2,914 times where the fixer fields are reassigned under this combination.

The feature combinations are also good indicators for other types of field reassignment and refinement. For example, in Netbeans, we notice that certain combinations of reporter, component, and assignee are good indicators for priority refinement. In the collected dataset of Netbeans, the combination of reporter ="soldatov", component ="code", and assignee

="issue" appear 166 times, and there is a total of 120 times that the priority fields are reassigned under this combination

To simulate the decision process of an experienced bug triager who needs to decide what bug report fields will get reassigned and refined, we create a baseline approach that infers reassigned and refined fields based on the statistics of the top-10 discriminative features for each field reassignment and refinement. Since there are two types of features, i.e., meta features and textual features, we compute the statistics for these two types of features in different ways.

For a meta feature m, the baseline approach computes the probability for a field f to be reassigned and refined as follows: given a value v of the meta feature m, suppose the number of times v appears in the training bug reports is denoted as $total$, and the number of times v appears in the training bug reports whose field f get reassigned and refined is denoted as $reassign$, the probability that field f of a bug report with value v of meta feature f, to get reassigned, which is denoted by $prob_{f,t}$, and is computed by $\frac{reassign}{total}$.

For a textual feature t, and for a field f, we compute the number of bug reports in the training data whose field f is reassigned or refined (denoted as $total_f$), and also the number of bug reports in the training data whose field f is reassigned or refined and contains term t (denoted as $reassign_{f,t}$). Based on the these numbers, we compute the probability for f to be reassigned given textual feature t, which is denoted as $prob_{f,t}$, by taking the ratio of $reassign_{f,t}$ and $total_f$.

To predict whether a field f will get reassigned and refined in a new bug report, the baseline approach first computes a probability for each of the top-10 discriminative features, considering the values of these features in the new bug report. If one of the probabilities is larger than or equal to 0.5, the baseline approach predicts that field f will get reassigned and refined; else it predicts that f will not be reassigned or refined.

Table X presents the F-measure scores of Im-ML.KNN compared with the baseline approach. We improve the average F-measure of the baseline approach by 7.32%, 28.71%, 29.87%, and 51.06% for OpenOffice, Netbeans, Eclipse, and Mozilla, respectively. Averaging across the four datasets, the average improvement achieved by Im-ML.KNN is 29.24%.

H. RQ5: Varying the Amount of Training Data

In the previous research questions, we use 10-fold cross validation, which means that 90% of the bug reports are used as training data. In this research questions, we would like to investigate the impact of reducing the amount of training data on the evaluation of our approach. In K-fold cross validation, $\frac{k-1}{k} * 100\%$ of the data is used to train a model, and the remaining $\frac{1}{k} * 100\%$ of the data is used to test the model. K-fold cross validation is a standard approach. To answer this research question, we try to reduce the number of folds to reduce the amount of training data. We vary the number of K from 2 to 10, and for each value of K, we perform 10 times K-fold cross-validation, and record the average F-measure scores.

Figure 7 presents the average F-measure scores of Im-ML.KNN for OpenOffice, Netbeans, Eclipse, and Mozilla with various amount of training bug reports. The average F-measure scores vary from 0.4541 – 0.6226, 0.5624 – 0.5963, 0.5296 – 0.5683, and 0.5652 – 0.5956, for OpenOffice, Netbeans, Eclipse, and Mozilla, respectively. When we vary K from 10 to 2, the F-measure for Eclipse, Mozilla, and Firefox remains relatively stable (it fluctuates less than 5.68% of the original value). For OpenOffice, the F-measure reduces by 26.81% when we vary K from 10 to 2.

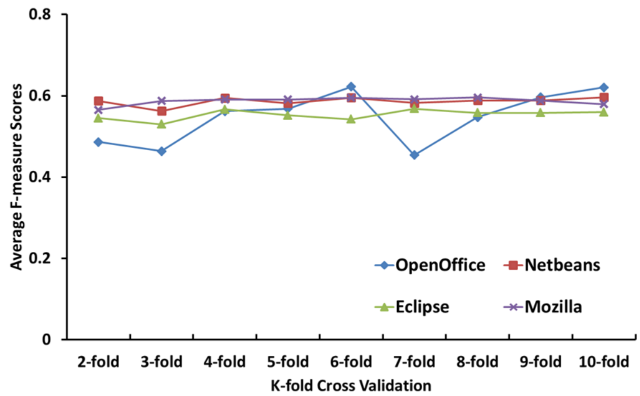

Fig. 7. Average F-measure of Im-ML.KNN for OpenOffice, Netbeans, Eclipse, and Mozilla with various amount of training bug reports.

I. RQ6: Time Efficiency of Im-ML.KNN

The time efficiency of Im-ML.KNN may affect its usability, therefore, we also investigate Im-ML.KNN's time efficiency. In this question, we investigate whether the runtime of Im-ML.KNN is reasonable. To answer this research question, we investigate the average amount of time that is needed by Im-ML.KNN and the baseline approaches to process a bug report during the model building phase, and the average amount of time it needs to predict the fields which would get reassigned and refined for a new bug report during the prediction phase. We use a Windows 7, 64-bit, Intel(R) Xeon(R) 2.53GHz server with 24GB RAM.

Table XI shows the average model building time and prediction time per bug report for Im-ML.KNN, Lamkanfi el al.'s approach, ML.KNN, and HOMER-NB. We notice that the average model building time and the average prediction time of Im-ML.KNN are 0.0265 and 0.0158 seconds per bug report, respectively. Comparing with the other 3 baseline approaches, the average model building time and the average prediction time of Im-ML.KNN are better than those of Lamkanfi el al.'s approach and HOMER-NB, and worse than those of ML.KNN. Still, the time taken by Im-ML.KNN is reasonable. Note that the model building phase can be done offline (e.g., overnight). Also, a learned model can be used to predict reassigned and refined fields of many new bug reports, it is normal to spend a few hours to process hundreds of thousands of bug reports to initially build a model, since the model can be reused and the process can be done offline.

VII. Discussion

A. Longitudinal Data Setup

To investigate whether Im-ML.KNN can be used to solve the problem in the same setting as the one in practice,

TABLE X
EXPERIMENT RESULTS OF *Im-ML.KNN* COMPARED WITH THE BASELINE APPROACH.

Projects	Algorithms	Ave.F-mea	Comp.F-mea.	Pro.F-mea.	Sev.F-mea.	Pri.F-mea.	OS F-mea.	Ver.F-mea.	Fix.F-mea.	Stat.F-mea.
OpenOffice	Im-ML.KNN	0.6204	0.7284	0.7944	0.7551	0.2954	0.3583	0.6522	0.9056	0.4742
	Baseline	0.5781	0.6913	0.7756	0.8848	0.1140	0.1001	0.6863	0.8553	0.5374
	Improvement.	**7.32%**	**5.37%**	**2.43%**	**-12.68%**	**159.20%**	**257.82%**	**-4.97%**	**5.88%**	**-11.76%**
Netbeans	Im-ML.KNN	0.5963	0.9304	0.8693	0.0000	0.2856	0.5306	0.5821	0.7788	0.1971
	Baseline	0.4633	0.7878	0.7208	0.0000	0.0970	0.0852	0.8223	0.6840	0.0460
	Improvement.	**28.71%**	**18.09%**	**20.60%**	**0.00%**	**194.53%**	**522.87%**	**-29.21%**	**13.86%**	**328.54%**
Eclipse	Im-ML.KNN	0.5597	0.6365	0.7334	0.2577	0.5413	0.6606	0.6341	0.8667	0.1475
	Baseline	0.4310	0.5992	0.6862	0.0760	0.4890	0.2175	0.5459	0.7866	0.0438
	Improvement.	**29.87%**	**6.23%**	**6.88%**	**239.28%**	**10.69%**	**203.68%**	**15.37%**	**10.19%**	**236.74%**
Mozilla	Im-ML.KNN	0.5796	0.7395	0.8123	0.2392	0.4817	0.7301	0.5843	0.8685	0.1813
	Baseline	0.3837	0.6585	0.6776	0.0461	0.2631	0.2169	0.3875	0.7989	0.0211
	Improvement.	**51.06%**	**12.31%**	**19.88%**	**419.14%**	**83.08%**	**236.68%**	**50.80%**	**8.71%**	**760.84%**

TABLE XI
AVERAGE MODEL BUILDING TIME, AND PREDICTION TIME, PER BUG REPORT, FOR IM-ML.KNN, LAMKANFI EL AL.'S APPROACH, ML.KNN, AND HOMER-NB (IN SECONDS).

Projects	Model Building Time				Prediction			
	Im.ML-KNN	Lamkanfi el al.'s	ML.KNN	HOMER-NB	Im.ML-KNN	Lamkanfi el al.'s	ML.KNN	HOMER-NB
OpenOffice	0.0270	0.0333	0.0043	0.0299	0.0136	0.0330	0.0045	0.0317
Netbeans	0.0263	0.0348	0.0046	0.0391	0.0153	0.0344	0.0046	0.0296
Eclipse	0.0291	0.0443	0.0059	0.0382	0.0182	0.0435	0.0061	0.0389
Mozilla	0.0234	0.0384	0.0050	0.0346	0.0162	0.0420	0.0053	0.0348
Average.	**0.0265**	**0.0377**	**0.0050**	**0.0355**	**0.0158**	**0.0382**	**0.0051**	**0.0338**

we performed an experiment using a longitudinal data setup following Tamrawi et al. and Bhattacharya and Neamtiu [18], [33]. We sorted the bug reports in the order they are received (i.e., temporally) and split them into 11 non-overlapping time windows of equal sizes, numbered 0 to 10. The process then proceeds as follows: First, in fold 1, we train using bug reports in window 0, and test the trained model using the bug reports in window 1. Then, in fold 2, we train using bug reports in window 1, and test the trained model using the bug reports in window 2, and so on. We proceed in a similar manner for the next folds. In the final fold (i.e., fold 10), we train using bug reports in window 9, and test using the bug reports in window 10. We record the average performance across the 10 folds.

Table XII presents the average F-measure of Im-ML.KNN compared with Lamkanfi el al.'s, ML.KNN, and HOMER-NB. The average F-measure of *Im-ML.KNN* are 0.4187, 0.4848, 0.4154, and 0.4708 for OpenOffice, Netbeans, Eclipse, and Mozilla, respectively. The improvement of our method over Lamkanfi *et al.*'s method is substantial. We improve the average F-measure of the method proposed by Lamkanfi *et al.* by 54.20%, 53.17%, 73.87%, and 61.57% for OpenOffice, Netbeans, Eclipse, and Mozilla, respectively. Averaging across the four datasets, the average improvement achieved by *Im-ML.KNN* is 60.70%.

The improvement of our method over the ML.KNN method is substantial for Netbeans, Eclipse, and Mozilla. We improve the average F-measure of ML.KNN by 6.68%, 25.43%, and 17.25%, and 11.12% for OpenOffice, Netbeans, Eclipse, and Mozilla, respectively. Averaging across the four datasets, the average improvement achieved by *Im-ML.KNN* is 15.12%.

The improvement of our method over the HOMER-NB method is substantial for Netbeans, Eclipse, and Mozilla. We improve the average F-measure of HOMER-NB by 63.94%,

TABLE XII
AVERAGE F-MEASURE OF IM-ML.KNN COMPARED WITH LAMKANFI EL AL.'S, ML.KNN, AND HOMER-NB USING A LONGITUDINAL DATA SETUP. THE LAST ROW SHOW THE AVERAGE PERFORMANCE ACROSS THE 4 PROJECTS.

Projects	Im-ML.KNN	Lamkanfi el al.'s	Improvement.
OpenOffice	0.4187	0.2715	**54.20%**
Netbeans	0.4848	0.3165	**53.17%**
Eclipse	0.4154	0.2389	**73.87%**
Mozilla	0.4708	0.2914	**61.57%**
Average.	**0.4474**	**0.2796**	**60.70%**

Projects	Im-ML.KNN	ML.KNN	Improvement.
OpenOffice	0.4187	0.3925	**6.68%**
Netbeans	0.4848	0.3865	**25.43%**
Eclipse	0.4154	0.3543	**17.25%**
Mozilla	0.4708	0.4237	**11.12%**
Average.	**0.4474**	**0.3892**	**15.12%**

Projects	Im-ML.KNN	HOMER-NB	Improvement.
OpenOffice	0.4187	0.2554	**63.94%**
Netbeans	0.4848	0.2836	**70.90%**
Eclipse	0.4154	0.2218	**87.31%**
Mozilla	0.4708	0.2761	**70.56%**
Average.	**0.4474**	**0.2592**	**73.18%**

70.90%, 87.31%, and 70.56% for OpenOffice, Netbeans, Eclipse, and Mozilla, respectively. Averaging across the four datasets, the average improvement achieved by *Im-ML.KNN* is 73.18%.

Figure 8, 9, 10, and 11 present the average F-measure scores of Im-ML.KNN, Lamkanfi el al.'s approach, ML.KNN, and HOMER-NB for OpenOffice, Netbeans, Eclipse, and Mozilla, respectively. We note that the average F-measure scores of Im-ML.KNN outperform those of other approaches for most folds for the majority of the projects. Also, in OpenOffice,

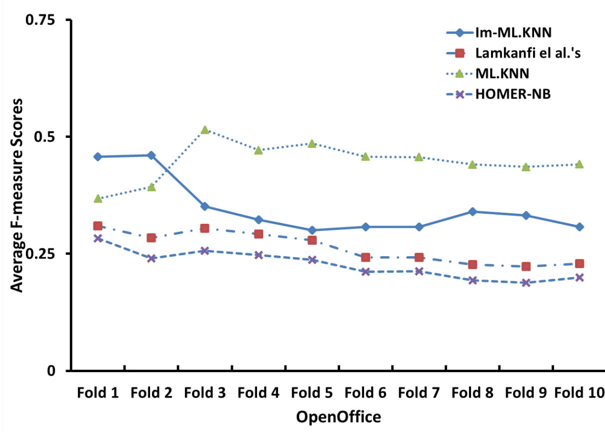

Fig. 8. Experiment Results of Im-ML.KNN, Lamkanfi el al.'s approach, ML.KNN, and HOMER-NB for OpenOffice with different folds.

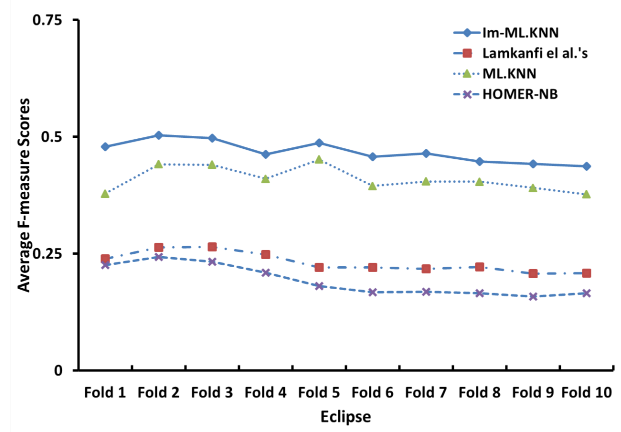

Fig. 10. Experiment Results of Im-ML.KNN, Lamkanfi el al.'s approach, ML.KNN, and HOMER-NB for Eclipse with different folds.

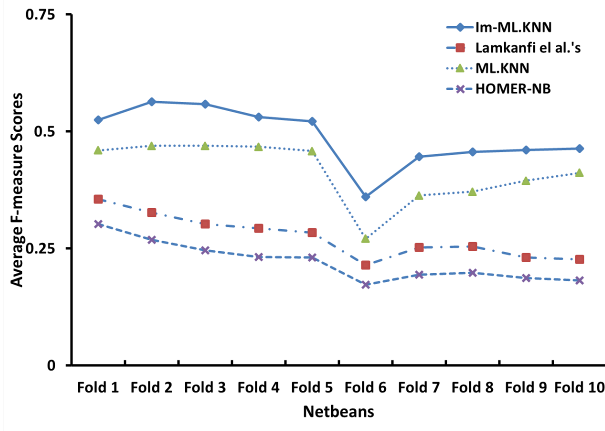

Fig. 9. Experiment Results of Im-ML.KNN, Lamkanfi el al.'s approach, ML.KNN, and HOMER-NB for netbeans with different folds.

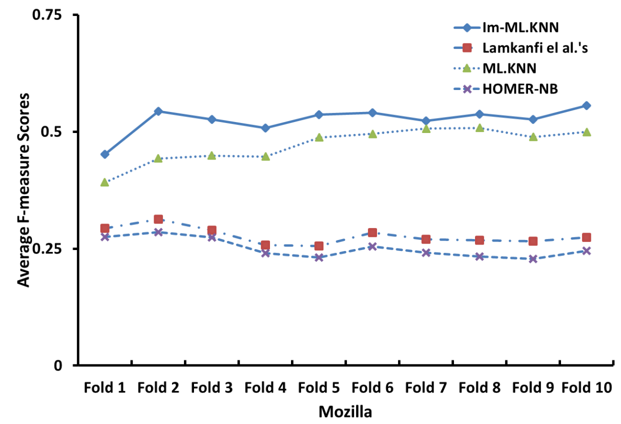

Fig. 11. Experiment Results of Im-ML.KNN, Lamkanfi el al.'s approach, ML.KNN, and HOMER-NB for Mozilla with different folds.

we notice there is a remarkable drop on Folds 8 and 9 for Im-ML.KNN, we double check the results, and it is the case.

B. Cost Analysis

Here, we analyze the cost of Im.ML-KNN. For a field f, if we correctly predict that the field will get reassigned and refined, let us assume that the time saved for bug fixing is s_f; else if we wrongly predict that the field would get reassigned and refined, we assume that the extra time needed for bug fixing is w_f. Suppose we have n bug reports, and among the n bug reports, we correctly predict whether field f would get reassigned in m bug reports and incorrectly predict whether f would get reassigned in k bug reports, then the cost for Im-ML.KNN is $(m \times s_f) - (k \times w_f)$. If $(k \times w_f) \leq (m \times s_f)$, then our approach could help developers save time.

Table XIII presents the cost of Im-ML.KNN. We denote the time saved due to the correct prediction of component, product, severity, priority, os, version, fixer, and status reassignments and refinements as $s_c, s_p, s_s, s_{pri}, s_o, s_v, s_f$, and s_{st}, and the time wasted due to wrong prediction as $w_c, w_p, w_s, w_{pri}, w_o, w_v, w_f$, and w_{st}. For simplicity, let us assume that the time saved for each correct prediction is the same, i.e., $s_c = s_p = s_s = s_{pri} = s_o = s_v = s_f = s_{st} = s$, and the time wasted for each wrong prediction is the same, i.e., $w_c = w_p = w_s = w_{pri} = w_o = w_v = w_f = w_{st} = w$. Then, the net savings for OpenOffice, Netbeans, Eclipse, and Mozilla are $298045 \times s - 39307 \times w$, $297323 \times s - 45092 \times w$, $358063 \times s - 47049 \times w$, and $269939 \times s - 141301 \times w$, respectively. For OpenOffice, Netbeans, Eclipse, and Mozilla, when $s > 0.1319 \times w$, $s > 0.1614 \times w$, $s > 0.1314 \times w$, and $s > 0.5235 \times w$, respectively, Im-ML.KNN could help save bug fixing time.

C. Impact of Default Assignment in Fixer Field

When a bug report is initially submitted, its fixer field could be assigned to a default address. For example, in OpenOffice, the fixer fields of some bug reports are set to "issues" when they are first submitted. It is easy to know that the fixer fields set to a default address will eventually get reassigned. Thus, we would like to investigate the effectiveness of Im-ML.KNN when we omit these default developer assignments.

To do so, we first remove bug reports whose fixers are initially set to default addresses. In OpenOffice and Netbeans, we remove bug reports whose fixers are initially set to default

TABLE XIII
COST OF IM-ML.KNN.

Projects	Component	Product	Severity	Priority
OpenOffice	$38883 \times s_c - 3286 \times w_c$	$39649 \times s_p - 2520 \times w_p$	$41988 \times s_s - 181 \times w_s$	$33544 \times s_{pri} - 8625 \times w_{pri}$
Netbeans	$42232 \times s_c - 4113 \times w_c$	$42624 \times s_p - 3721 \times w_p$	0	$34001 \times s_{pri} - 12344 \times w_{pri}$
Eclipse	$43917 \times s_o - 6722 \times w_o$	$48251 \times s_v - 2388 \times w_v$	$42327 \times s_s - 8312 \times w_s$	$45257 \times s_{pri} - 5382 \times w_{pri}$
Mozilla	$31085 \times s_c - 20320 \times w_c$	$28524 \times s_p - 22881 \times w_p$	$37982 \times s_s - 13423 \times w_s$	$31016 \times s_{pri} - 20389 \times w_{pri}$

Projects	OS	Version	Fixer	Status
OpenOffice	$38615 \times s_o - 3554 \times w_o$	$39019 \times s_v - 3150 \times w_v$	$36044 \times s_f - 6125 \times w_f$	$30303 \times s_{st} - 11866 \times w_{st}$
Netbeans	$44471 \times s_o - 1874 \times w_o$	$44302 \times s_v - 2043 \times w_v$	$35671 \times s_f - 10674 \times w_f$	$36022 \times s_{st} - 10323 \times w_{st}$
Eclipse	$49174 \times s_o - 1465 \times w_o$	$46390 \times s_v - 4249 \times w_v$	$41531 \times s_f - 9108 \times w_f$	$41216 \times s_{st} - 9423 \times w_{st}$
Mozilla	$30298 \times s_o - 21107 \times w_o$	$34582 \times s_v - 16823 \times w_v$	$38780 \times s_f - 12625 \times w_f$	$37673 \times s_{st} - 13732 \times w_{st}$

TABLE XIV
AVERAGE F-MEASURE OF IM-ML.KNN COMPARED WITH IM-ML.KNN^{Default} USING 10 TIMES 10-FOLD CROSS VALIDATION SETUP. THE LAST ROW SHOW THE AVERAGE PERFORMANCE ACROSS THE 4 PROJECTS.

Projects	Im-ML.KNN	Im-ML.KNN^{Default}
OpenOffice	0.6204	0.5038
Netbeans	0.5963	0.5555
Eclipse	0.5597	0.5965
Mozilla	0.5796	0.5279
Average.	**0.5890**	**0.5459**

TABLE XV
AVERAGE F-MEASURE OF IM-ML.KNN COMPARED WITH IM-ML.KNN^{Default} USING A LONGITUDINAL DATA SETUP. THE LAST ROW SHOW THE AVERAGE PERFORMANCE ACROSS THE 4 PROJECTS.

Projects	Im-ML.KNN	Im-ML.KNN^{Default}
OpenOffice	0.4187	0.4494
Netbeans	0.4848	0.4565
Eclipse	0.4154	0.4036
Mozilla	0.4708	0.3254
Average.	**0.4474**	**0.4087**

addresses such as “issues”, “UNKNOWN”, “spreadsheet”, and “support”. In Eclipse, we remove bug reports whose fixers are initially set to default addresses which end with “inbox”. In Mozilla, we remove bug reports whose fixers are initially set to default addresses such as “nobody”, “timeless”, “accounts”, and “bugzilla”. In total, we have 35,934, 32,938, 12,801, and 11,416 bug reports remaining for OpenOffice, Netbeans, Eclipse, and Mozilla, respectively. We run Im-ML.KNN on these datasets, and denote the resultant results as Im-ML.KNN^{Default}.

Table XIV and XV present the average F-measure of Im-ML.KNN compared with Im-ML.KNN^{Default} using 10 times 10-fold cross validation setup, and the longitudinal data setup, respectively. On average across the 4 projects, Im-ML.KNN^{Default} achieves average F-measures 0.5459 and 0.4087 in the cross validation setup and longitudinal data setup respectively. We notice the average F-measures of Im-ML.KNN^{Default}, but the differences are relatively small (i.e., < 0.05).

From the intution, removing the bug reports whose fixer fields are set to a default address may remove the bug reports submitted by users, and leave the reports that were created by developers, since the developers are the ones with the project knowledge to assign to something other than default. We also investigate whether it is the case. For example, in OpenOffice, we find a developer Frank Schonheit has reported 1,150 bug reports, and fixed 146 bug reports. But still 7 out of the 1,150 reported bugs are assigned to the default address “issue”. Also, a user deye only has reported 1 bug report 102816, but the fixer field is set to “ab@bregas.de” initially. Thus, users and developers in the community all have the chance to set the fixer field to a default address.

D. Evaluation

In this paper, we automatically identify good α, β, γ, and $threshold_l$ values for *Im-ML.KNN* following the algorithm presented in Figure 1. The values are optimized (and thus are different) for different datasets and different training frames in our 10-fold cross validation and longitudinal data setup.

Also, for each label l, which corresponds to a type of bug report field reassignment, our *Im-ML.KNN* automatically identify good $threshold_l$ from the training bug reports. Thus, the decision boundaries (aka. thresholds) of *Im-ML.KNN* are fixed, i.e., the precision and recall would not be varied with different threshold values. We believe our *Im-ML.KNN* could help developers to use our tool in practice, since they do not need to consider the effect of the threshold values to the performance of our tool.

In single-label learning literature, receiver operating characteristic (ROC) curve is a graphical plot which shows the performance of a binary classifier when its threshold is varied [34]. Notice in our paper, we do not analyze the ROC curve due to 3 reasons:

- *Im-ML.KNN* automatically identifies good thresholds for each of the 8 types of field reassignment, which makes the thresholds fixed. Thus, there is only one point in the ROC curve for each of the 8 types of field reassignment.
- In multi-label learning literature, ROC curve is rarely used to evaluate the performance of a multi-label learning algorithm, c.f., [13], [26]–[28]. Normally, researchers prefer to use precision, recall, and F-measure to measure the performance of a multi-label learning algorithm.
- In our paper, there are in total 8 types of field reassignment, which correspond to 8 labels. If for each label, we plot its ROC curve, then there would be too many curves.

TABLE XVI
AVERAGE F-MEASURE OF IM-ML.KNN COMPARED WITH HOMER-KNN USING 10-FOLD CROSS VALIDATION SETUP. THE LAST ROW SHOW THE AVERAGE PERFORMANCE ACROSS THE 4 PROJECTS.

Projects	Im-ML.KNN	HOMER-KNN	Impovement.
OpenOffice	0.6204	0.5974	**3.85%**
Netbeans	0.5963	0.5409	**10.24%**
Eclipse	0.5597	0.5309	**4.44%**
Mozilla	0.5796	0.5667	**2.28%**
Average.	**0.5890**	**0.5602**	**5.20%**

TABLE XVII
AVERAGE F-MEASURE OF IM-ML.KNN COMPARED WITH HOMER-KNN USING A LONGITUDINAL DATA SETUP. THE LAST ROW SHOW THE AVERAGE PERFORMANCE ACROSS THE 4 PROJECTS.

Projects	Im-ML.KNN	HOMER-KNN	Improvement.
OpenOffice	0.4187	0.3851	**8.71%**
Netbeans	0.4848	0.4300	**12.73%**
Eclipse	0.4154	0.3905	**6.37%**
Mozilla	0.4708	0.4245	**10.93%**
Average.	**0.4474**	**0.4075**	**9.68%**

E. Im-ML.KNN vs. HOMER-KNN

In the previous section, we use Naive Bayes as the underlying classifier for HOMER, which is also used by Tsoumakas et al. [15]. We notice HOMER could also use other underlying classifiers. Thus, we also use kNN as the underlying classifier of HOMER (denoted as HOMER-KNN) as we do in *Im-ML.KNN*, and we set the number of neighbors in kNN as 10 as *Im-ML.KNN*.

Table XVI and XVII present the average F-measure of Im-ML.KNN compared with HOMMER-KNN using 10 times 10-fold cross validation setup, and longitudinal data setup, respectively. From Table XVI, we note that the improvement of our method over HOMER-KNN is substantial. We improve the average F-measure of HOMER-KNN by 3.85%, 10.24%, 4.44%, and 2.28% for OpenOffice, Netbeans, Eclipse, and Mozilla, respectively. Averaging across the four datasets, the average improvement achieved by *Im-ML.KNN* is 5.20% in the 10-fold cross validation setup. From Table XVII, We improve the average F-measure of HOMER-KNN by 8.71%, 12.73%, 6.37%, and 10.93% for OpenOffice, Netbeans, Eclipse, and Mozilla, respectively. Averaging across the four datasets, the average improvement achieved by *Im-ML.KNN* is 9.68% in the longitudinal data setup.

F. Qualitative Analysis

Here, we perform a qualitative analysis on why the features that we consider are relevant, how our *Im-ML.KNN* could potentially help reduce bug fixing time, and why we need to combine the 3 classifiers. We take status reassignment and refinement (i.e., a bug report is reopened) and component reassignment and refinementas examples.

Features: Table XVIII, XIX, XX and XXI present 4 bug reports from Netbeans. These 4 bug reports are chose from the prediction results of our Im-ML.KNN, Lamkanfi el al.'s, ML.KNN, and HOMER-NB. Also, since our Im-ML.KNN is a nearest neighbor based approach, it would be easy to explain why our approach outperforms the baseline approaches if some fields of the bug reports are similar. Note that we record the values of fields in these bug reports when they are first reported - before any reassignments and refinements (if any). The status and component fields in these 4 bug reports get reassigned and refined. We notice there are many similarities among these 4 bug reports:

- Many values of meta features in these 4 bug reports are the same. For example, the product, component, assignee,

TABLE XVIII
BUG REPORT #137543 OF NETBEANS.

Summary:	Red information message in ”New Database Connection” dialog
Description:	According http://wiki.netbeans.org/InformationIcons 1. Start IDE. Open ”Services” window. 2. Rightclick ”Databases” node and choose ”New Connection...”. Result: ”New Database Connection” dialog appears with ”Please specify a value for the required field Database:” message in the bottom. This message is RED and looks like error message. View will be better if message is not red (probably just black).
Product:	db
Component:	UI
Reporter:	Roman Mostyka
Assignee:	davidvc
Version:	6.X
Priority:	P4
Platform:	ALL
OS:	Windows XP

TABLE XIX
BUG REPORT #137600 OF NETBEANS.

Summary:	I18N - some driver names in New Connection dialog drop-down not localized
Description:	using new db connection dialog, the word Host does not come from the pseudo localized bundle file - the word host is in these bundles, but its not coming from the localized bundle as is all the other labels. this label is in bundle file org-netbeans-modules-db/NewConnectionHost =&Host:
Product:	db
Component:	UI
Reporter:	Ken Frank
Assignee:	davidvc
Version:	6.X
Priority:	P2
Platform:	ALL
OS:	Sun

TABLE XX
BUG REPORT #144020 OF NETBEANS.

Summary:	Ugly New Database connection dialog UI
Description:	Please, fix all UI issues in New Database connection dialog like, incorrect alignment of UI components, the size of components, spacing, ... See the attached screenshot.
Product:	db
Component:	UI
Reporter:	Petr Blaha
Assignee:	davidvc
Version:	6.X
Priority:	P3
Platform:	ALL
OS:	ALL

TABLE XXI
BUG REPORT #149041 OF NETBEANS.

Summary:	Wrong size for some fields in created table
Description:	1. Connect to ”travel” Java DB. 2. Rightclick ”Tables” and choose ”Create Table...”. 3. Set table name, column name, choose INTEGER type, set size to 3 and press ”OK”. Result: Table with column is created, but size of column is 10, not 3 as it was set. I guess it can be true not only for INTEGER type.
Product:	db
Component:	UI
Reporter:	Roman Mostyka
Assignee:	davidvc
Version:	6.X
Priority:	P3
Platform:	ALL
OS:	ALL
Creation Date:	2008-10-03
Fixed Date:	2009-05-12

and version fields for these 4 bug reports are “db”, “UI”, “davidvc”, and “6.X”, respectively. Also, for bug report #137543 and #149041, their reporter are the same, i.e., “Roman Mostyka”.

- The description of these 4 bug reports are similar, i.e., their textual features are similar. These 4 bug reports are all about UI issues and they share some common terms, such as “dialog” and ”connection”.

From the observation, we notice that if we consider both the meta and textual features, the performance of *Im-ML.KNN* could be further improved.

Reducing Bug Fixing Time: Notice that bug report #149041 in Table XXI takes a long time to be fixed. It is created in “2008-10-03”, and is only fixed in ”2009-05-12”. The bug fixing time is more than half a year. The component and status fields of this bug report are reassigned and refined. These reassignments and refinements mean that the component that this bug affects is wrongly reported (which leads to the component reassignment) and the bug is initially fixed incorrectly and needs to be re-fixed (which leads to status reassignment). These reassignments and refinements are likely to increase bug fix time. If our system is deployed, it can predict that the component field is likely to be reassigned and refined, and the bug report is likely to be reopened (which implies that the bug is hard-to-fix). This can guide developers to first find the right component before attempting to fix the bug, and to be more careful in performing the fix, as the fix is likely to be a risky one. As a result, the bug fixing time is likely to be reduced. Note that the model building time for our *Im-ML.KNN* is only several hours at most (and it could be trained offline), and the typical prediction time for a bug report is less than a second. Thus, from developers' point of view, they can get alert information in just less than a second.

Composing Classifiers: Note that we have 3 classifiers in *Im-ML.KNN*, and the prediction results for the 3 classifiers could be different. The combination of these 3 classifiers could utilize the advantages of each classifier, and achieve a better performance. For example, if we predict the fields which would get reassigned and refined for bug report #149041, we notice that the meta features for #149041 are very similar to the other 3 bug reports (#137543, #137600, and #144020), however its textual features are less similar to the other 3 bug reports. Thus, the meta classifier predicts that the component and status fields would get reassigned and refined; on the other hand, the textual classifier does not predict any of these 2 fields would get reassigned and refined, while the mixed classifier predicts that only the component field would get reassigned and refined. By combining these 3 classifiers, *Im-ML.KNN* predicts that the component and status fields of bug report #149041 would get reassigned and refined.

VIII. THREATS TO VALIDITY

In this section, we highlight threats to internal validity, external validity, and construct validity.

Threats to internal validity. Threats to internal validity relate to bias and errors in our experiments. To reduce the risk of this threat, we have double checked our datasets and our experiments, however there could still be errors that we did not notice. Also, to reduce training set selection bias, we have applied 10-fold cross-validation 10 times, and recorded the average performance. And we have also evaluated Im-ML.KNN using the longitudinal data setup.

Threats to external validity. Threats to external validity relate to the generalizability of our results. To reduce this risk, we have analyzed 190,558 closed and fixed bug reports from 4 open source software projects, and investigate 8 types of bug report field reassignment. Analyzing a substantial proportion of bug reports in selected projects is important for the generalizability of the findings. Past studies also only investigate similar number of bug reports from these projects [33], [35]–[38]. In the future, we plan to reduce this threat further by analyzing more bug reports from more software projects, including commercial and open source projects.

Threats to construct validity. Threats to construct validity refer to the suitability of our evaluation measures. We use the average F-measure score as the main evaluation metric which is also used by past studies to evaluate the effectiveness of a prediction technique in various software engineering studies [20], [22], [31], [39]. Thus, we believe there is little threat to construct validity.

IX. RELATED WORK

In this section, we briefly review studies on bug report field reassignments in Section IX-A, and multi-label learning in software engineering in Section IX-B.

A. Bug Report Field Reassignment

The most related work to our paper is the empirical study we perform [9]. In the empirical study, we analyze the root causes of bug report field reassignment by sending emails to developers in open source software projects. We also analyze various field reassignments that happen in 190,558 bug reports in 4 open-source software projects. We find that approximately 80% of the bug reports have one or more of their fields reassigned, and the bug reports whose fields get reassigned require

more time to be fixed than those without field reassignments. This work complements our previous work, and our previous work serves as a motivation to this work. In particular, in this paper we propose an automated tool to predict which bug report fields would get reassigned, to help developers reduce bug fixing effort.

There have been a number of other studies on bug report field reassignments. Guo *et al.* perform an empirical study on fixer reassignments, and they find five primary reasons for fixer reassignments, i.e., difficulty to identify the root cause, ambiguous ownership of components, poor bug report quality, difficulty to determine the proper fix, and workload balancing [40]. Jeong *et al.* investigate fixer reassignments in Mozilla and Eclipse, and they propose a method which uses fixer reassignment graph to improve the performance of bug triaging [7]. Bhattacharya *el at.* extend Jeong *et al.'s* work to improve the accuracy of bug triaging by using multi-feature fixer reassignment graph [33]. Shihab *et al.* study reopened bugs in Eclipse, Apache HTTP, and OpenOffice, and find that the average time to resolve a reopened bug is approximately twice as much as the time to resolve a non-reopened bug [4], [5]. They propose a machine learning based method to predict reopened bug reports; they extract 4 groups of features, related to work habits, bug report fields, bug fix, and people, containing a total of 24 features. Sureka investigates the component reassignment problem in Eclipse and Mozilla, and proposes the use of machine learning algorithms to predict the components of bug reports [12]. Lamkanfi *et al.* also study the component reassignment problem, and find that the proportion of bug reports whose component field gets reassigned varies between 8.3% to 32.7% in Eclipse and Mozilla [6]. They propose the usage of Naive Bayes to predict whether the component of a bug report would be reassigned, and their method achieves precision and recall between 0.58-0.94 and 0.54-1 for bug reports of several products of Eclipse and Mozilla. Our work generalizes the above studies; whereas previous studies focus on single bug report field reassignment, our work considers all field reassignments simultaneously.

B. Multi-Label Learning

There have been a number of studies on multi-label learning in software engineering [20], [21], [36], [41]. Xia et al. propose *TagCombine* to recommend tags in software information sites [21]. Xia et al. propose *DevRec* to recommend bug resolvers [36]. Each of these two studies makes use of a multi-label learning algorithm. Banerjee et al. propose the usage of multi-label learning algorithms to select suitable duplicated bug report detection techniques, and combine them to achieve a better performance [41]. Xia el al. propose a composite method *MLL-GA* which combines different multi-label learning algorithms by leveraging genetic algorithms, to achieve a better performance for software behavior learning [20]. Our work is orthogonal to the above studies since we study a different problem - we focus on predicting which bug report fields would get reassigned rather than recommending a set of tags, resolvers, and duplicated bug report detection techniques, and predicting the fault types of a failure. Also, different from the above studies, in this study we consider the class imbalance problem, and adapt ML.KNN to handle this problem.

X. Conclusion and Future Work

In this paper, we develop a tool which leverages *multi-label learning* algorithms to automatically predict which *bug report fields would be reassigned and refined*. We propose imbalanced ML.KNN (*Im-ML.KNN*), which extends ML.KNN, by considering the class imbalance phenomenon. *Im-ML.KNN* is a composite model, which combines 3 multi-label classifiers built on different types of features (i.e., meta, textual, and mixed features). We evaluate the performance of *Im-ML.KNN* on 4 large-scale open source projects which contain 190,558 bug reports in total. Experiment results show that *Im-ML.KNN* could achieve an average F-measure score of 0.56-0.62. We also compare *Im-ML.KNN* with other state-of-art methods, such as the method proposed by Lamkanfi *et al.*, ML.KNN, and HOMER. The results show that *Im-ML.KNN* on average improves the average F-measure scores of Lamkanfi *et al.*'s method, *ML.KNN*, and HOMER-NB by 119.69%, 9.11%, and 161.08%, respectively. We show that the performance of *Im-ML.KNN* remains relatively stable across a wide range of parameter settings thus showing that it is not sensitive on the optimal setting of its parameter.

In the future, we plan to evaluate *Im-ML.KNN* with more bug reports from more software projects, and develop a better technique which could improve the bug report field reassignment and refinementprediction further.

Acknowledgment

This research was supported by the National Basic Research Program of China (the 973 Program) under grant 2015CB352201, National Key Technology R&D Program of the Ministry of Science and Technology of China under grant 2014BAH24F02, and the Fundamental Research Funds for the Central Universities.

References

[1] M. Newman, "Software errors cost us economy $59.5 billion annually," *NIST Assesses Technical Needs of Industry to Improve Software-Testing*, 2002.

[2] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss, "What makes a good bug report?" *Software Engineering, IEEE Transactions on*, vol. 36, no. 5, pp. 618–643, 2010.

[3] T. Zimmermann, R. Premraj, J. Sillito, and S. Breu, "Improving bug tracking systems," in *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*. IEEE, 2009, pp. 247–250.

[4] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Predicting re-opened bugs: A case study on the eclipse project," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 2010, pp. 249–258.

[5] ——, "Studying re-opened bugs in open source software," *Empirical Software Engineering*, pp. 1–38, 2012.

[6] A. Lamkanfi and S. Demeyer, "Predicting reassignments of bug reports-an exploratory investigation," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 2013, pp. 327–330.

[7] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 111–120.

[8] R. K. Saha, S. Khurshid, and D. E. Perry, "An empirical study of long lived bugs," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 144–153.
[9] X. Xia, D. Lo, M. Wen, E. Shihab, and B. Zhou, "An empirical study of bug report field reassignment," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 174–183.
[10] S. Mani, S. Nagar, D. Mukherjee, R. Narayanam, V. S. Sinha, and A. A. Nanavati, "Bug resolution catalysts: identifying essential non-committers from bug repositories," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 193–202.
[11] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *International Journal of Data Warehousing and Mining (IJDWM)*, vol. 3, no. 3, pp. 1–13, 2007.
[12] A. Sureka, "Learning to classify bug reports into components," in *Objects, Models, Components, Patterns*. Springer, 2012, pp. 288–303.
[13] M.-L. Zhang and Z.-H. Zhou, "Ml-knn: A lazy learning approach to multi-label learning," *Pattern Recognition*, vol. 40, no. 7, pp. 2038–2048, 2007.
[14] H. He and E. A. Garcia, "Learning from imbalanced data," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 21, no. 9, pp. 1263–1284, 2009.
[15] G. Tsoumakas, I. Katakis, and I. Vlahavas, "Effective and efficient multilabel classification in domains with large number of labels," in *Proc. ECML/PKDD 2008 Workshop on Mining Multidimensional Data (MMD?8)*, 2008, pp. 30–44.
[16] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 361–370.
[17] R. A. Baeza-Yates and B. A. Ribeiro-Neto, *Modern Information Retrieval*, 2011.
[18] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 365–375.
[19] J. Han, M. Kamber, and J. Pei, *Data mining: concepts and techniques*. Morgan kaufmann, 2006.
[20] X. Xia, Y. Feng, D. Lo, Z. Chen, and X. Wang, "Towards more accurate multi-label software behavior learning," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 134–143.
[21] X. Xia, D. Lo, X. Wang, and B. Zhou, "Tag recommendation in software information sites," in *Proceedings of the Tenth International Workshop on Mining Software Repositories*. IEEE Press, 2013, pp. 287–296.
[22] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 386–396.
[23] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE, 2008, pp. 346–355.
[24] F. Thung, D. Lo, and J. L. Lawall, "Automated library recommendation," in *WCRE*, 2013, pp. 182–191.
[25] G. Tsoumakas, E. Spyromitros-Xioufis, J. Vilcek, and I. Vlahavas, "Mulan: A java library for multi-label learning," *Journal of Machine Learning Research*, 2011.
[26] G. Tsoumakas, I. Katakis, and L. Vlahavas, "Random k-labelsets for multilabel classification," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 23, no. 7, pp. 1079–1089, 2011.
[27] J. Read, B. Pfahringer, G. Holmes, and E. Frank, "Classifier chains for multi-label classification," *Machine learning*, vol. 85, no. 3, pp. 333–359, 2011.
[28] M. Zhang and Z. Zhou, "A review on multi-label learning algorithms," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 26, no. 8, pp. 1819–1837, Aug 2014.
[29] Y. Feng and Z. Chen, "Multi-label software behavior learning," in *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 2012, pp. 1305–1308.
[30] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 15–25.
[31] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Multi-layered approach for recovering links between bug reports and fixes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 63.
[32] F. Peters and T. Menzies, "Privacy and utility for defect prediction: Experiments with morph," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 189–199.
[33] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–10.
[34] T. Fawcett, "An introduction to roc analysis," *Pattern recognition letters*, vol. 27, no. 8, pp. 861–874, 2006.
[35] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 34–43.
[36] X. Xia, D. Lo, X. Wang, and B. Zhou, "Accurate developer recommendation for bug resolution," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 2013, pp. 72–81.
[37] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: a case study on firefox," in *Proceedings of the 8th working conference on mining software repositories*. ACM, 2011, pp. 93–102.
[38] X. Xia, D. Lo, E. Shihab, X. Wang, and B. Zhou, "Automatic, high accuracy prediction of reopened bugs," *Automated Software Engineering*, pp. 1–35, 2014.
[39] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 382–391.
[40] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Not my bug! and other reasons for software bug report reassignments," in *Proceedings of the ACM 2011 conference on Computer supported cooperative work*. ACM, 2011, pp. 395–404.
[41] S. Banerjee, Z. J. Syed, J. Helmick, and C. Bojan, "A fusion approach for classifying duplicate problem reports," in *ISSRE*, 2013, pp. 208–217.

Xin Xia received his PhD degree from the College of Computer Science and Technology, Zhejiang University, China in 2014. He is currently a research assistant professor in the college of computer science and technology at Zhejiang University. His research interests include software analytic, empirical study, and mining software repository. He is a member of the Institute of Electrical and Electronics Engineers.

David Lo received his PhD degree from the School of Computing, National University of Singapore in 2008. He is currently an assistant professor in the School of Information Systems, Singapore Management University. He has close to 10 years of experience in software engineering and data mining research and has more than 130 publications in these areas. He received the Lee Foundation Fellow for Research Excellence from the Singapore Management University in 2009. He has won a number of research awards including an ACM distinguished paper award for his work on bug report management. He has published in many top international conferences in software engineering, programming languages, data mining and databases, including ICSE, FSE, ASE, PLDI, KDD, WSDM, TKDE, ICDE, and VLDB. He has also served on the program committees of ICSE, ASE, KDD, VLDB, and many others. He is a steering committee member of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER) which is a merger of the two major conferences in software engineering, namely CSMR and WCRE. He will also serve as the general chair of ASE 2016. He is a leading researcher in the emerging field of software analytics and has been invited to give keynote speeches and lectures on the topic in many venues, such as the 2010 Workshop on Mining Unstructured Data, the 2013 Gnie Logiciel Empirique Workshop, the 2014 International Summer School on Leading Edge Software Engineering, and the 2014 Estonian Summer School in Computer and Systems Science.

Emad Shihab is an Assistant Professor in the Department of Computer Science and Software Engineering at Concordia University. He received his PhD from Queen's University and his MASc. and BEng. from the University of Victoria. Dr. Shihab's research interests are in Software Engineering, Software Quality Assurance, Empirical Software Engineering, Mining Software Repositories, Mobile Applications and Software Architecture. He worked as a software research intern at Research In Motion in Waterloo, Ontario and Microsoft Research in Redmond, Washington. He held an NSERC Alexander Graham Bell Canada Graduate Scholarship (CGS-D3) and the PhD research achievement award from the School of Computing at Queens University. He served as organizer to a number of events related to Mining Software Repositories (MSR) and Mobile Applications, including serving as program chair of the MSR 2012 Challenge Track and the MSR 2013 Data Showcase Track. Dr. Shihab regularly serves on the programming committee of Software Engineering conferences and journals such as ICSME, MSR, ICPC, SANER (formerly WCRE/CSMR), OSS, TSE and EMSE.

Xinyu Wang received the bachelors and PhD degrees in computer science from Zhejiang University of China, in 2002 and 2007. He was a research assistant in Zhejiang University, during 2002-2007. He is currently an associate professor in the College of Computer Science, Zhejiang University. His research interests include software engineering, formal methods, and very large information systems.