10-2017

Which packages would be affected by this bug report?

Qiao HUANG

David LO
Singapore Management University, davidlo@smu.edu.sg

Xin XIA

Qingye WANG

Shanping LI

DOI: https://doi.org/10.1109/ISSRE.2017.24

2017 IEEE 28th International Symposium on Software Reliability Engineering

Which Packages Would be Affected by This Bug Report?

Qiao Huang*, David Lo†, Xin Xia‡✓, Qingye Wang*, and Shanping Li*
*College of Computer Science and Technology, Zhejiang University, China
†School of Information Systems, Singapore Management University, Singapore
‡Department of Computer Science, University of British Columbia, Canada
{tkdsheep, wqyy, shan}@zju.edu.cn, davidlo@smu.edu.sg, xxia02@cs.ubc.ca

***Abstract*—A large project (e.g., Ubuntu) usually contains a large number of software packages. Sometimes the same bug report in such project would affect multiple packages, and developers of different packages need to collaborate with one another to fix the bug. Unfortunately, the total number of packages involved in a project like Ubuntu is relatively large, which makes it time-consuming to manually identify packages that are affected by a bug report. In this paper, we propose an approach named *PkgRec* that consists of 2 components: a name matching component and an ensemble learning component. In the name matching component, we assign a confidence score for a package if it is mentioned by a bug report. In the ensemble learning component, we divide the training dataset into *n* subsets and build a sub-classifier on each subset. Then we automatically determine an appropriate weight for each sub-classifier and combine them to predict the confidence score of a package being affected by a new bug report. Finally, *PkgRec* combines the name matching component and the ensemble learning component to assign a final confidence score to each potential package. A list of top-k packages with the highest confidence scores would then be recommended. We evaluate *PkgRec* on 3 datasets including Ubuntu, OpenStack, and GNOME with a total number of 42,094 bug reports. We show that *PkgRec* could achieve recall@5 and recall@10 scores of 0.511-0.737, and 0.614-0.785, respectively. We also compare *PkgRec* with other state-of-art approaches, namely LDA-KL and MLkNN. The experiment results show that *PkgRec* on average improves recall@5 and recall@10 scores of LDA-KL by 47% and 31%, and MLkNN by 52% and 37%, respectively.**

***Index Terms*—Bug Report, Package Recommendation, Multi-Label Classification**

I. Introduction

During software development and maintenance, bugs are inevitable and bug fixing is a time-consuming and costly task. Many software projects use bug tracking systems (e.g., Bugzilla and JIRA) to manage bug reporting, bug resolution, and bug archiving processes [1].

A large project (e.g., Ubuntu) usually contains a large number of software packages. Sometimes the same bug report in such project would affect multiple packages, and developers of different packages need to collaborate with one another to fix the bug. Notice that when we say a bug report affects a package, it means that developers need to release a patch for the package (i.e., change its code) to fix the bug.

Since the total number of packages involved in a project like Ubuntu is relatively large (e.g., in our dataset, after removing inactive packages, we still have 341 packages in Ubuntu), it is time-consuming to manually identify packages that would be affected by a bug report. Thus, in this paper, we are interested in developing an automated approach to process a new bug report and recommend a list of software packages[1] that are possibly affected by this bug report. We denote this problem as *package recommendation for bug resolution* (or package recommendation, for short). Once a bug report is received, recommending suitable packages that are likely to be affected can reduce the time and cost of the bug fixing process.

In this paper, We propose an automatic approach named *PkgRec* that consists of 2 components: name matching component and ensemble learning component. The name matching component is based on the observation: some bug reports may mention the full names (or part of the names) of several packages in the title or description, and these packages are likely to be affected. Thus, we assign a confidence score for a package if it is mentioned by a bug report. Notice that the name matching component does not work well if no packages are mentioned by a bug report or the packages mentioned are not affected by the bug report. To deal with the limitation of the first component, we create another component, which performs text classification on the textual contents of a bug report, to recommend potentially affected packages. This component, referred to as the ensemble learning component, divides the training dataset into *n* subsets and build a sub-classifier on each subset. Then we automatically determine an appropriate weight for each sub-classifier and combine them to predict the confidence score of a package being affected by a new bug report. *PkgRec* combines name matching component and ensemble learning component to assign a final confidence score to each potential package. A list of top-k packages with the highest scores would then be recommended. Our experiment results show that the combination of these two components would improve the overall performance.

We evaluate our approach on 3 datasets: Ubuntu[2], OpenStack[3] and GNOME[4]. In total, we analyze 42,094 bug reports.

✓Corresponding author.

[1]In this paper, we also refer the third-party projects that a bug affects as ”package” since they are used in a project such as Ubuntu in the same way as third-party packages.
[2]https://bugs.launchpad.net/ubuntu
[3]https://bugs.launchpad.net/openstack
[4]https://bugs.launchpad.net/gnome

We measure the effectiveness of our approach in terms of recall@5 and recall@10 following previous studies in software engineering [2]–[4]. For the 3 datasets, our approach can achieve recall@5 and recall@10 scores of up to 0.741, and 0.785 respectively. We compare our approach with 2 state-of-the-art approaches, namely LDA-KL [5] and MLkNN [6]. Our approach on average improves recall@5 and recall@10 scores of LDA-KL by 47.25% and 31.41%, and MLkNN by 52.49% and 37.36%, respectively.

The main contributions of this paper are:

- We propose *PkgRec* to automatically recommend packages that are possibly affected by a bug report.
- We evaluate *PkgRec* on 3 datasets with 42,094 bug reports in total. The experiment results show that *PkgRec* outperforms LDA-KL and MLkNN by a statistically significant margin.

Paper organization. The remainder of this paper is organized as follows. Section II presents the motivation and preliminaries of our approach *PkgRec*. Section III elaborates on the details of *PkgRec*. Section IV and Section V present the experiment setup and results on 3 datasets. Section VI discusses other aspects of *PkgRec*, and threats to validity. Section VII surveys the related work. Finally, Section VIII concludes the paper and points out potential future directions.

II. Preliminaries

In this section, we first present a motivating example of bug reports affecting multiple packages. Then we introduce the preliminary materials, including bug report representation and multi-label classification.

Motivating Example. Table I presents an example of bug report in Ubuntu project that affects multiple packages. The bug report has a field called *Affects*, which records the packages marked by developers for further investigation. For each of the marked package, there is also a field called *status* to record whether it is truly affected. In this bug report, 2 packages are affected, namely *GTK+* and *unity-2d*. Note that different packages can have their own status for the same bug report. In our motivating example, the nautilus package is also in the *Affects* list, but its status is *Invalid* (i.e., not truly affected). We only consider a package as truly affected if its status is *Fix Released*.

TABLE I
Bug Report #804435 in Ubuntu Project

Bug ID: #804435 **Summary:** Wallpaper is loaded twice with different alignment by gnome-session and nautilus (Oneiric) **Affects:** nautilus (Status: Invalid) GTK+ (Status: Fix Released) unity-2d (Status: Fix Released) **Bug Description:** When using Unity-2D and gnome-session-fallback in Oneiric the wallpaper, painted by nautilus, is not loaded correctly at session startup. For some seconds is not aligned with screen, there is a large left margin colored grey, then after some seconds is reloaded correctly and well-aligned. ProblemType: Bug DistroRelease: Ubuntu 11.10 Package: nautilus 1:3.1.2-0ubuntu2 ProcVersionSignature: Ubuntu 3.0-2.3-generic 3.0.0-rc4 Uname: Linux 3.0-2-generic i686 Architecture: i386 Date: Fri Jul 1 18:51:10 2011 ProcEnviron: PATH=(custom, no user) LANG=it_IT.UTF-8 SHELL=/bin/bash SourcePackage: nautilus UpgradeStatus: No upgrade log present (probably fresh install)

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

1) The bug report describes a wallpaper loading problem in *nautilus* package, but it affects *unity-2d* and *GTK+*.
2) In the bug report description, 3 packages are mentioned, namely *unity-2d*, *gnome-session-fallback* and *nautilus*. This a good indicator for us to automatically find packages that are possibly affected by the bug report. However, only *unity-2d* is truly affected by this bug report, while the other 2 packages are not affected (e.g., the final status of the bug report for *nautilus* is "Invalid").
3) By manually reading the developers' discussion, we find that in the early stage of bug fixing, *nautilus* is one of the suspected packages. For example, one developer said in the comment: "*I confirm this bug, and it really seems to be caused by nautilus*." However, 2 months later, another developer confirmed that it is a bug in *GTK+*. Note that *GTK+* is a package for creating graphical user interfaces, which shares common features with packages like *unity-2d*. Thus, simple name matching is not sufficient to find affected packages given a bug report.

Bug Report Representation. In this paper, we use the Vector Space Model (VSM) [7] to represent each bug report as a vector of feature values. In this model, a feature can be viewed as a dimension, and a bug report can then be viewed as a data point in a high-dimensional space. We extract words from the summary and description texts as features. More specifically, for each bug report, we concatenate the summary and description text, then tokenize the text, remove stop words, stem them (i.e., reduces them to their root forms, e.g., tests and testing are reduced to test) by using Porter stemmer[5], and represents them in the form of a vector. We ignore the text of developer discussion since it is not available at the time a new bug report is submitted. Finally, to calculate the weight of each feature (i.e., word), we use TF-IDF [8], which is widely used as a weighting factor in information retrieval and text mining. To measure the similarity of two bug reports, we use cosine similarity, which is widely used to calculate text similarity in information retrieval and text mining domains [9], [10].

Multi-Label Classification. Given a data instance (i.e., bug report), the task of multi-label classification is to predict a set of labels (i.e., packages) that should be assigned to it. Standard classification task only assigns one label to each data instance.

[5]http://tartarus.org/martin/PorterStemmer/

However, in many settings a data instance can be assigned to more than one label. In our work, each data instance (i.e.,a bug report) can also be assigned with multiple labels (i.e., packages).

MLkNN is a state-of-the-art algorithm in the multi-label classification literature [6]. To infer the labels for a new instance (i.e., bug report) X_{new}, MLkNN follows three steps: the computation of membership counting scores, the computation of MLkNN confidence scores, and the assignment of labels. We describe the details of each step in the following paragraphs.

Membership Counting Score. For a new instance X_{new}, M-LkNN first finds its k-nearest neighbors $knn(X_{new})$ from the training dataset. Then for each label (i.e., package) *l* in the label set *L*, it counts the number of instances in $knn(X_{new})$ that are assigned to label *l*, denoted as $C_{X_{new}}(l)$.

MLkNN Confidence 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 that 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 $knn(X_{new})$. Then, the MLkNN confidence score for *l* and X_{new} is the probability that the X_{new} is assigned to *l*, given that exactly $C_{X_{new}}(l)$ instances in $knn(X_{new})$ are assigned to label *l*. Formally, we have the following equation:

$$ML(X_{new}, l) = P(H_1^l \mid E^l_{C_{X_{new}}(l)}) \quad (1)$$

From Equation 1, and using Bayes rule, we can derive:

$$ML(X_{new}, H_1^l) = \frac{P(H_1^l) \times P(E^l_{C_{X_{new}}(l)} \mid H_1^l)}{\sum_{i\in\{0,1\}} P(H_i^l) \times P(E^l_{C_{X_{new}}(l)} \mid H_i^l)} \quad (2)$$

The parameters of $P(H_1^l)$, $P(H_0^l)$, $P(E_m^l \mid H_1^l)$ and $P(E_m^l \mid H_0^l)$ can be inferred from the training dataset. The detail of the inference process is available in [6].

Label Assignment. In MLkNN, if the confidence score of H_1^l is larger than that of H_0^l, then label *l* would be assigned to X_{new}. In this paper, instead of outputting predicted labels for X_{new}, we modify MLkNN to recommend the top-k labels that have the highest confidence scores, denoted as $MLkNN(X_{new}, H_1^l)$.

III. Approach

In this section, we propose *PkgRec*, an automatic approach to recommend a list of software packages that are possibly affected by a bug report. *PkgRec* consists of 2 components: name matching component and ensemble learning component. In Section III-A and Section III-B, we present technical details of the name matching component and ensemble learning component, respectively. In Section III-C we present a composition of these two components that would result in *PkgRec*.

A. Name Matching Component

From Table I, we notice that some packages are mentioned in the description of the bug report. We observe this phenomenon by manually reading a large number of bug reports. Thus, we guess that if the title or description of a bug report contains the name of a package, then the bug described in the report has a relatively high probability of affecting the package. Many packages use compound words as their names. For example, the "gdk-pixbuf" package has 2 words in its name: gdk and pixbuf. In practice, our name matching component also considers the situation that part of the name of a package is mentioned in the title or description of a bug report.

Given a bug report *b* and a package *p*, the name matching component would assign for the bug report *b* a confidence score *Name(b, p)* that denotes the likelihood of this bug report *b* to affect package *p*. The confidence score *Name(b, p)* is given by the following equation:

$$Name(b,p) = \begin{cases} 1 & \text{completely matched} \\ \frac{nameHits(b,p)}{wordsInName(p)} & \text{otherwise} \end{cases} \quad (3)$$

The equation considers two situations. If the full name of a package *p* is mentioned (i.e., completely matched) in the title or description of a bug report *b*, then the confidence score of package *p* being affected by bug report *b* is set to 1. Otherwise, we first tokenize the package name into a set of single words, and count the number of single words in the package name, denoted as $wordsInName(p)$. Then we count the number of single words that appear in the title or description of a bug report, denoted as $nameHits(b, p)$. Finally, the confidence score is calculated as the ratio of single words being mentioned by a bug report. For example, for package "xserver-xorg-video-intel", if only the word "intel" appears in the bug report, then the confidence score would be 1/4 = 0.25.

Note that if a bug report does not mention any package, then name matching component cannot recommend any packages. Also, the packages mentioned in the bug report may not be affected. Thus, we design another component based on multi-label classification to get more accurate results.

B. Ensemble Learning Component

Many bug reports do not mention any package name in their titles or descriptions, making the name matching component useless. To leverage all textual features in a bug report, we design the ensemble learning component, which builds multi-label classifiers (i.e., MLkNN) on the training bug reports.

Previous studies [11], [12] have shown that ensemble learning, which trains multiple classifiers (with either random initialization and/or different subsets of the training set) and combine them, can help to overcome overfitting problem. In the ensemble learning component, instead of simply building one MLkNN classifier on the whole training dataset, we divide the training dataset into n equal-sized disjoint sets and build n sub-classifiers (i.e., MLkNN).

When we combine these sub-classifiers, we also assign a weight factor to each of them. Ideally, if a sub-classifier performs well on the testing dataset, then it should be assigned with a higher weight factor. However, we cannot know the true labels of testing dataset when we determine the value of weight factors in the training phase. So we build a mock set of bug reports for preliminary testing. That is, for each bug report in the real testing dataset, we find its "nearest neighbor" bug report in the training dataset, and add this "nearest neighbor" into the mock set. Thus, the feature distribution of bug reports in the mock set should be relatively similar to those in real testing dataset. Then we evaluate each sub-classifier on the mock set, and determine the weight factors according to their performance on the mock set.

Formally, we calculate the values of a vector $\vec{\alpha} = \{\alpha_1, \alpha_2, ..., \alpha_n\}$, where α_i denotes the weight factor of the i^{th} sub-classifier C_i. The value of α_i is given by the following equation:

$$\alpha_i = \frac{EC(C_i, MockSet)}{\sum_{1\leqslant j\leqslant n} EC(C_j, Mockset)} \tag{4}$$

In the above equation, $EC(C_i, MockSet)$ is the performance of the i^{th} sub-classifier C_i on the mock set when using a certain evaluation criterion *EC*. By default, we set the evaluation criterion *EC* as recall@k (see Section V).

Finally, given a new bug report *b* and a package *p*, the ensemble learning component would combine the *n* sub-classifiers to calculate a confidence score of package *p* being affected by bug report *b*, denoted as $Ensemble(b, p)$, which is given by the following equation:

$$Ensemble(b, p) = \sum_{i=1}^{n} \alpha_i \times C_i(b, p) \tag{5}$$

In the above equation, $C_i(b, p)$ denotes the confidence score of package *p* being affected by bug report *b* when applying the i^{th} sub-classifier on the test case.

Algorithm 1 presents the pseudo-code to estimate appropriate weight values of $\vec{\alpha}$. We first divide the training bug reports into n equal-sized disjoint sets and build a sub-classifier on each subset (Lines 9 and 10). Specifically, we apply MLkNN to build these sub-classifiers. Then we create a mock set of bug reports from the training set that are similar to the bug reports in testing set. To do so, for each bug report in testing set, we find its nearest bug report in training set using cosine similarity and add this nearest neighbor into the mock set (Lines 11-15). After that, we evaluate each sub-classifier C_i on mock set and calculate the score of the given evaluation criterion *EC*, to determine the weight factor α_i (Line 16-19). Finally, we return $\vec{\alpha}$ (Line 20).

Algorithm 1 *EstimateWeights*: Estimation of $\vec{\alpha}$ in ensemble learning component

```
1: EstimateWeights(TrainSet, TestSet, EC)
2: Input:
3: TrainSet: Training set of bug reports
4: TestSet: Testing set of bug reports
5: EC: Evaluation criterion
6: Output:
7: $\vec{\alpha}$: Weight vector for the sub-classifiers
8: Method:
9: Divide the training bug reports into n equal-sized disjoint sets;
10: Built subset classifiers $C_1$, $C_2$, . . . , $C_n$ on the n disjoint sets;
11: MockSet = ∅;
12: for all bug report br ∈ $TestSet$ do
13:     Find its nearest bug report $br'$ in TrainSet using cosine
        similarity;
14:     Add $br'$ into MockSet;
15: end for
16: for i from 1 to n do
17:     Evaluate $C_i$ on MockSet and calculate the score of EC;
18:     Compute $\alpha_i$ according to Equation 4;
19: end for
20: return $\vec{\alpha}$;
```

C. PkgRec: A Composite Approach

As shown in previous sections, given a bug report *b* and a package *p*, we can get confidence score $Name(b, p)$ and $Ensemble(b, p)$ from the name matching component and ensemble learning component, respectively. In this section, we propose *PkgRec* which combines both $Name(b, p)$ and $Ensemble(b, p)$ to calculate a composite confidence score $PkgRec(b, p)$, as follows:

$$PkgRec(b, p) = \gamma_1 * Name(b, p) + \gamma_2 * Ensemble(b, p) \tag{6}$$

Where γ_1, γ_2 ∈ [0,1] represent the weight factors of name matching score and ensemble learning score to the overall *PkgRec* score. Similar to Algorithm 1, the values of γ_1 and γ_2 are also automatically determined by separately evaluating the name matching component and ensemble learning component on the mock set.

Finally, for a new bug report, *PkgRec* would recommend the top-k packages that have the highest confidence scores.

IV. Experiment Setup

In this section, we describe the experiment setup that we follow to evaluate the performance of our approach. We evaluate *PkgRec* on 3 datasets and compare it with LDA-KL and MLkNN. The experimental environment is a computer equipped with Intel(R) Core(TM) i5-2410M CPU and 4GB RAM, running Windows 7 (64-bit).

A. Dataset and Experiment Settings

We collect our datasets from 3 open source projects: Ubuntu, OpenStack and GNOME. Table II presents the statistics of our datasets. The columns correspond to the project name (Project), the time period of collected bug reports (Time), the number of collected reports (# Reports), the number of unique features (i.e., words) in the collected reports (# Terms), the number of packages (# Packages) and the average number of packages that a bug report affects (# Avg. Affects).

For each dataset, we delete the words which appear in less than 0.1% of all bug reports (e.g., in GNOME dataset, a term

TABLE II
STATISTICS OF COLLECTED BUG REPORTS.

Project	Time	# Reports	# Features	# Packages	# Avg. Affects
Ubuntu	2004-12-26 - 2011-11-29	18,530	3,808	341	2.005
Openstack	2012-05-16 - 2014-10-09	18,207	2,839	71	1.203
GNOME	2005-06-18 - 2016-05-26	5,357	3,016	167	1.757

is removed if it appears in less than 5 bug reports). Inspired by Al-Kofahi et al.'s work [13], we also remove inactive packages because they may introduce noise. In practice, we can use historical data to guide the removal of such packages. Specifically, we remove packages that are affected by less than 0.1% of all bug reports (e.g., in GNOME dataset, a package is removed if it is affected by less than 5 bug reports) since such packages are practically inactive.

To simulate the usage of our approach in practice, we use the same longitudinal data setup described in [5], [14]. For each dataset, we sort the bug reports in chronological order of creation time, and then divide them into 10 non-overlapping folds (or windows) of equal sizes. The evaluation process proceeds as follows: First, we use bug reports in fold 1 as training data, and test the bug reports in fold 2. Then, we use bug reports in 2 folds (i.e., fold 1 and fold 2) as training data, and test using the bug reports in fold 3, and so on. In the final fold, we train using bug reports in fold 1 to 9, and test using bug reports in fold 10.

There are 2 parameters in *PkgRec*: the number of nearest neighbors *n* in MLkNN, and the number of sub-classifiers *k* in ensemble learning component. In our experiment, both *k* and *n* are set to 10 by default. We also investigate the effect of varying these parameters. When using MLkNN alone as baseline, we also set *n* to 10 by default. We use Mulan[6] as the MLkNN implementation.

LDA-KL in [15] was first proposed for recommending components affected by a bug report. We can use LDA-KL for package recommendation too – by viewing packages as components. For the number of topics and iterations in LDA-KL, we use the same parameter setting as [15]. We use JGibbsLDA[7], which uses Gibbs sampling process, as the LDA implementation. More specifically, since the values of hyper-parameters (alpha and beta) in LDA were not given in [15], we use the recommended parameter setting in JGibbsLDA (i.e., alpha is 50/*K* and beta is 0.1, where *K* represents the number of topics). Finally, to enable others to use our techniques, we have published our source code and dataset on GitHub[8].

B. Evaluation Metrics

We evaluate the performance of *PkgRec* and other baseline approaches using two metrics: recall@k, and precision@k. The definitions of recall@k and precision@k are as follows:

Suppose that there are *m* bug reports. For each bug report b_i, let the set of its actual affected packages be D_i. We recommend the set of top-k packages P_i for b_i using our approach (or the baselines). The recall@k and precision@k for the *m* bug reports are given by:

[6]http://mulan.sourceforge.net/
[7]http://jgibblda.sourceforge.net/
[8]http://github.com/tkdsheep/MultiPackage

TABLE III
CLIFF'S DELTA AND THE EFFECTIVENESS LEVEL

Cliff's Delta ($\mid\delta\mid$)	Effectiveness Level
$\mid\delta\mid < 0.147$	Negligible
$0.147 \le \mid\delta\mid < 0.33$	Small
$0.33 \le \mid\delta\mid < 0.474$	Medium
$\mid\delta\mid \ge 0.474$	Large

$$Recall@k = \frac{1}{m}\sum\frac{|P_i \bigcap D_i|}{|D_i|} \tag{7}$$

$$Precision@k = \frac{1}{m}\sum\frac{|P_i \bigcap D_i|}{|P_i|} \tag{8}$$

We focus on top-k since practitioners are not likely to check too many packages, c.f., [16]–[18]. Notice that our approach is meant to be a recommendation tool. For such setting, recall (ability to find affected packages) is more important than precision, c.f., [19], [20]. Thus, we focus on recall@k in our experiment, and the value of k is set to 5 and 10, which also follows previous software engineering studies [2]–[4]. However, we still discuss the precision@k of *PkgRec*, and present the details in Section VI.

V. EXPERIMENT RESULTS

A. RQ1: How effective is PkgRec? How much improvement can it achieve over other state-of-the-art approaches?

Motivation. To show that *PkgRec* is useful, one of the first questions is to see how effective it is in performing its package recommendation and whether it can perform as well as, or better than state-of-the-art approaches. Answering this research question would shed light on how much *PkgRec* advances the state-of-the-art.

Approach. To answer this research question, we compare *PkgRec* with 2 state-of-the-art approaches, namely LDA-KL and MLkNN. We record the average recall@5 and recall@10 across different folds of training data for each project.

To check if the differences in the performance of *PkgRec* and the baseline approaches are statistically significant, for each dataset, we run the Wilcoxon signed-rank test [21] at 95% significance level on two competing approaches. Since we run the test many times, we use Bonferroni correction [22] to counteract the results of multiple comparisons. We also compute Cliff's delta (δ) [23], which is a non-parametric effect size measure that quantifies the amount of difference between two approaches. The delta values range from -1 to 1, where δ = -1 or 1 indicates the absence of overlap between two approaches (i.e., all values of one group are higher than the values of the other group, and vice versa), while δ = 0 indicates the two approaches are completely overlapping. Table III describes the meaning of different Cliff's delta values and their corresponding interpretation [23].

TABLE IV
RECALL@5 AND RECALL@10 OF *PkgRec* AND LDA-KL, AND THE IMPROVEMENT OF *PkgRec* OVER LDA-KL (IMPROVE.). THE LAST ROW SHOWS THE AVERAGE RECALL@5 AND RECALL@10 SCORES OF *PkgRec* AND LDA-KL, AND THE AVERAGE IMPROVEMENT.

Projects	**Recall@5**					**Recall@10**				
	PkgRec	**LDA-KL**	**Improve.**	**p-value**	δ	**PkgRec**	**LDA-KL**	**Improve.**	**p-value**	δ
Ubuntu	0.511	0.290	76.21%	<0.01	0.85	0.614	0.388	58.25%	<0.01	0.88
OpenStack	0.679	0.584	16.27%	<0.05	0.73	0.784	0.703	11.52%	<0.05	0.75
GNOME	0.737	0.435	69.43%	<0.01	0.89	0.785	0.570	37.72%	<0.01	0.89
Average.	**0.642**	**0.436**	**47.25%**	-	-	**0.728**	**0.554**	**31.41%**	-	-

TABLE V
RECALL@5 AND RECALL@10 OF *PkgRec* AND MLKNN, AND THE IMPROVEMENT OF *PkgRec* OVER MLKNN (IMPROVE.).

Projects	**Recall@5**					**Recall@10**				
	PkgRec	**MLkNN**	**Improve.**	**p-value**	δ	**PkgRec**	**MLkNN**	**Improve.**	**p-value**	δ
Ubuntu	0.511	0.306	66.99%	<0.01	0.70	0.614	0.407	50.86%	<0.01	0.77
OpenStack	0.679	0.596	13.93%	<0.01	0.72	0.784	0.703	11.52%	<0.01	0.73
GNOME	0.737	0.360	104.72%	<0.01	0.89	0.785	0.479	63.88%	<0.01	0.89
Average.	**0.642**	**0.421**	**52.49%**	-	-	**0.728**	**0.530**	**37.36%**	-	-

Results. Table IV compares recall@5 and recall@10 of *P-kgRec* and LDA-KL. Table V compares recall@5 and recal-l@10 of *PkgRec* and MLkNN. The recall@5 and recall@10 of *PkgRec* vary from 0.511 to 0.737, and 0.614 to 0.785, respectively. The improvement of *PkgRec* over baseline approaches and the corresponding p-value of δ are also shown in the two tables. Notice that the "average" p-value and δ is not calculated because they are meaningless.

In each dataset, *PkgRec* outperforms both LDA-KL and MLkNN. From Table IV, *PkgRec* outperforms LDA-KL by 47.25% and 31.41% for average recall@5, and recall@10, respectively. In the Ubuntu dataset, *PkgRec* achieves the highest improvement of 76.21% and 58.25% over LDA-KL for recall@5 and recall@10, respectively. From Table V, *PkgRec* outperforms MLkNN by 52.49% and 37.36% for average recall@5 and recall@10, respectively. In the GNOME dataset, *PkgRec* achieves the highest improvement of 104.72% and 63.88% over MLkNN for recall@5 and recall@10, respectively.

We consider that *PkgRec* statistically significantly improves a baseline approach at the confidence level of 95% if the adjusted p-value is less than 0.05. Across the 3 datasets, every p-value is less than 0.05 and some of them are even less than 0.01. Also, δ varies from 0.72 to 0.89. Thus, *PkgRec* shows significant improvement over the baseline approaches with large effect size.

We also note that the performance of *PkgRec* (and also the baselines) varies between different projects. For example, the recall@5 of *PkgRec* on GNOME is approximately 40% higher than that on Ubuntu. One reason is that these projects are in different domains with different data distributions, which could impact the performance of *PkgRec*. Also, in RQ2, we find that the name matching component achieves relatively high recall on GNOME, which indicates that bug reports in GNOME are more likely to mention the name of the affected packages, thus making it easier for prediction.

B. RQ2: What is the performance of the ensemble learning component and name matching component?

Motivation. *PkgRec* has two components (i.e., ensemble learning component and name matching component) and we want to see if the combination of the two components results in better or poorer performance.

Approach. To answer this research question, we separately evaluate name matching component and ensemble learning component on the 3 datasets and compare their performance with that of *PkgRec*. Similar to RQ1, we run Wilcoxon signed-rank test with Bonferroni correction to check if the differences in the performance of *PkgRec* and each component of *PkgRec* are statistically significant. We also use Cliff's delta (δ) to measure the effect size of the difference between *PkgRec* and the 2 components.

Results. Table VI and Table VII present the recall@5 and recall@10 scores of *PkgRec* compared with those of ensemble learning component and name matching component. The improvement of PkgRec over the 2 components and the corresponding p-value of δ are also shown in the two tables.

In each dataset, *PkgRec* outperforms both the ensemble learning component and name matching component. Notice that the "contribution" of these two components to the performance of *PkgRec* may vary a lot in different datasets. For example, in OpenStack dataset, the major contribution comes from the ensemble learning component, while in GNOME dataset, name matching component contributes much more to the performance of *PkgRec*. On average of the three datasets, *PkgRec* outperforms the ensemble learning component by 44.59% and 29.08% for recall@5, and recall@10, respectively. *PkgRec* also outperforms the name matching component by 53.96% and 61.78% for average recall@5, and recall@10, respectively.

Across the 3 datasets, every p-value is less than 0.01 and δ varies from 0.62 to 0.89. Thus, *PkgRec* shows significant improvement over the 2 components with large effect size. The results show that it is beneficial to combine the ensemble learning component and name matching component.

Additionally, for each dataset, the performance of ensemble learning component also outperforms MLkNN baseline. For example, in the GNOME dataset, the recall@5 and recall@10 of ensemble learning component is 0.398 and 0.535, which improves MLkNN (its recall@5 and recall@10 are 0.360 and 0.479 respectively) by 10.56% and 11.69%, respectively.

TABLE VI
RECALL@5 AND RECALL@10 OF *PkgRec* AND ITS ENSEMBLE LEARNING COMPONENT.

Projects	Recall@5					Recall@10				
	PkgRec	**Ensemble.**	**Improve.**	**p-value**	δ	**PkgRec**	**Ensemble.**	**Improve.**	**p-value**	δ
Ubuntu	0.511	0.334	52.99%	<0.01	0.64	0.614	0.430	42.79%	<0.01	0.70
OpenStack	0.679	0.600	13.17%	<0.01	0.70	0.784	0.728	7.69%	<0.01	0.67
GNOME	0.737	0.398	85.18%	<0.01	0.89	0.785	0.535	46.72%	<0.01	0.89
Average.	**0.642**	**0.444**	**44.59%**	-	-	**0.728**	**0.564**	**29.08%**	-	-

TABLE VII
RECALL@5 AND RECALL@10 OF *PkgRec* AND ITS NAME MATCHING COMPONENT.

Projects	Recall@5					Recall@10				
	PkgRec	**Name.**	**Improve.**	**p-value**	δ	**PkgRec**	**Name.**	**Improve.**	**p-value**	δ
Ubuntu	0.511	0.361	41.55%	<0.01	0.83	0.614	0.407	50.86%	<0.01	0.85
OpenStack	0.679	0.232	192.67%	<0.01	0.89	0.784	0.261	200.38%	<0.01	0.89
GNOME	0.737	0.659	11.84%	<0.01	0.62	0.785	0.683	14.93%	<0.01	0.89
Average.	**0.642**	**0.417**	**53.96%**	-	-	**0.728**	**0.450**	**61.78%**	-	-

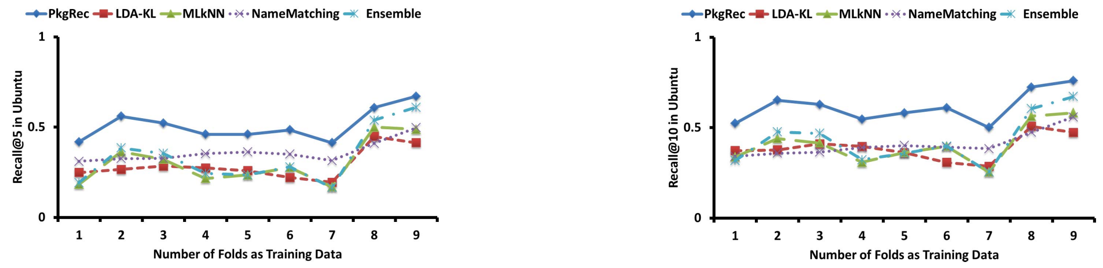


Fig. 1. Recall@5 and recall@10 for when using different number of folds as training data (Ubuntu dataset).

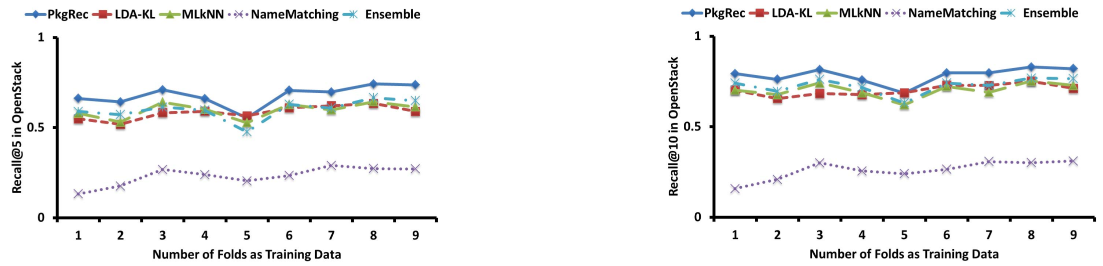


Fig. 2. Recall@5 and recall@10 for when using different number of folds as training data (OpenStack dataset).

C. RQ3: How effective is PkgRec when the amount of training data is varied?

Motivation. To evaluate the performance of *PkgRec*, we use the longitudinal data setup. The amount of training data available is different for different folds; the latter folds have more training data. In this research question, we investigate whether the performance of *PkgRec* increases when the amount of training data increases.

Approach. To answer this research question, we present the recall@5 and recall@10 scores when using different number of folds (from 1 fold to 9 folds) as training data.

Results. Figures 1, 2 and 3 present the recall@5 and recall@10 for *PkgRec* and other approaches with different amount of training data (fold 1 - fold 9). The results show that, for all of the folds, the recall@5 and recall@10 scores of *PkgRec* are always better than those of the other approaches.

We also notice that for several settings (such as using the first 5 folds as training data to test the 6^{th} fold in OpenStack dataset), the performance of *PkgRec* (and the other approaches) decreases as compared with other settings.

By manually investigating the data, we find that there are some completely new packages in the testing dataset which are never seen in the training dataset. For example, *Mistral* (a package to provide workflow service) and *Congress* (a package to provide policy as a service) are not affected by any bug reports in the first five folds of training data of OpenStack. Our approach (and the baselines) cannot recommend the new packages which affect its performance.

D. RQ4: How effective is PkgRec when varying the number of nearest neighbors in MLkNN?

Motivation. MLkNN is the underlying classifier of *PkgRec*. We need to set the parameter *k* (i.e., the number of nearest neighbors to compare) when using MLkNN. By default, *k* is set to 10. In this RQ, we investigate the performance of *PkgRec* with different settings of k. Answering this research question can help us identify a suitable range of parameter settings for MLkNN in *PkgRec*.

Approach. To answer this research question, we vary *k* from 1 to 30, with a step of 5. Note that in this experiment, the

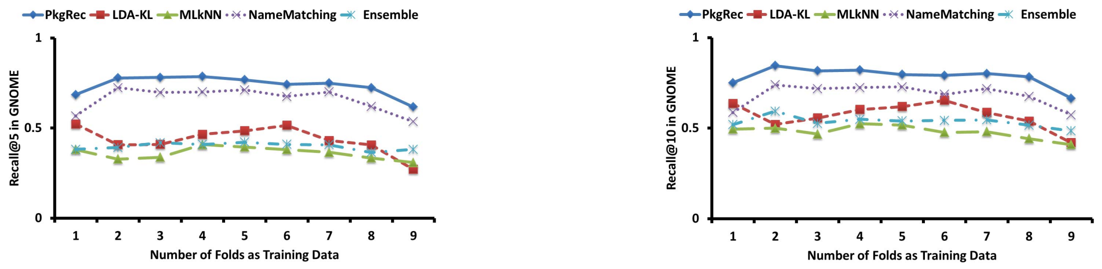


Fig. 3. Recall@5 and recall@10 for when using different number of folds as training data (GNOME dataset).

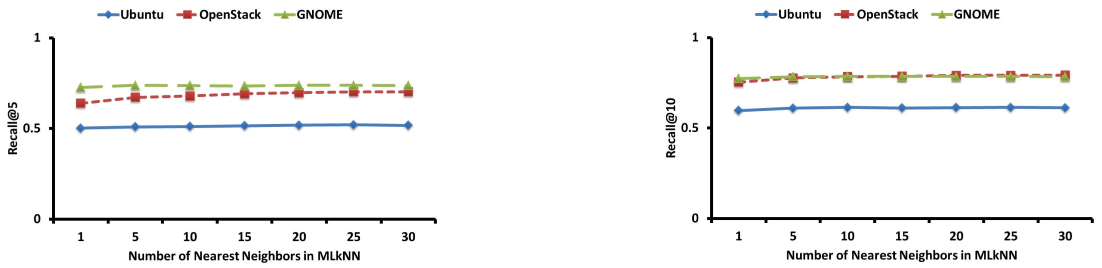


Fig. 4. Recall@5 and recall@10 of *PkgRec* on 3 datasets when varying the number of nearest neighbor in MLkNN

number of sub-classifiers in the ensemble learning component is fixed to 10.

Results. Figure 4 presents the recall@5 and recall@10 scores of *PkgRec* on 3 datasets when varying the number of nearest neighbor *k* in MLkNN. The results show that, in Ubuntu and GNOME dataset, the performance of *PkgRec* is generally stable across different settings of *k*. For example, in GNOME dataset, recall@5 and recall@10 scores vary from 0.727 to 0.738, and 0.774 to 0.785. In OpenStack dataset, setting a larger *k* can improve the performance of *PkgRec*. For example, recall@5 is 0.639 when k is set to 1, while recall@5 increases to 0.703 when k is set to 30, which corresponds to an improvement of 10.02%. In summary, our approach is robust with different parameter settings of *k*.

E. RQ5: How effective is PkgRec when the number of sub-classifiers in ensemble learning component is varied?

Motivation. By default, we build 10 sub-classifiers in the ensemble learning component. In this RQ, we also investigate the performance of PkgRec with a different number of sub-classifiers. Answering this research question can help us identify suitable parameter setting range for the ensemble learning component.

Approach. To answer this research question, we vary the number of sub-classifiers *n* from 6 to 15, with a step of 1. Note that in this experiment, the number of nearest neighbors in MLkNN is fixed to 10.

Results. Figure 5 presents recall@5 and recall@10 scores of *PkgRec* for the 3 datasets when varying the number of sub-classifier in ensemble learning component. The results show that the performance of *PkgRec* is generally stable across various numbers of sub-classifiers. For example, for OpenStack dataset, the recall@5 and recall@10 scores vary from 0.678 to 0.691, and 0.776 to 0.784 when the number of sub-classifiers is varied from 6 to 15. In summary, our approach is robust with different parameter settings of *n*.

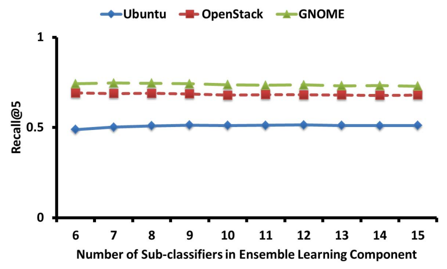
VI. Discussion

In this section, we discuss the precision@k of *PkgRec*, the time efficiency of *PkgRec* and threats to validity.

Precision@k of PkgRec. In RQ1, we focus on investigating recall@k of *PkgRec*. In this section, we also discuss the precision@k of *PkgRec*. Tables VIII and IX compare the precision@5 and precision@10 of *PkgRec*, LDA-KL, and MLkNN. The precision@5 and precision@10 of *PkgRec* vary from 0.152 to 0.257, and 0.089 to 0.136, respectively. These numbers might seem low. However, notice that the average number of packages affected by a bug report is low. Thus, the optimal precision@k value is also low. For example, in GNOME, the average number of packages affected by a bug report is 1.757. If we recommend top-10 packages, the best precision@10 would be 0.177. The precision@10 of *PkgRec* for the GNOME dataset is 0.136, which is close to the optimal value.

Also, the improvement of *PkgRec* over LDA-KL and M-LkNN on precision@k is substantial. Compared with LDA-KL, *PkgRec* improves the average precision@5, and precision@10 by 56.15% and 36.47%, respectively. Compared with MLkNN, *PkgRec* improves the average precision@5, and precision@10 by 61.11% and 43.21%, respectively. We also run Wilcoxon signed-rank test with Bonferroni correction and compute Cliff's delta. We find that the differences are all statistically significant (with p-values less than 0.05) and substantial (with Cliff's deltas $\geq$ 0.74).

Time Efficiency of PkgRec. The time efficiency of *PkgRec* will affect its practical use. Thus, we report the model building and prediction time of *PkgRec*, and compare them with those of LDA-KL and MLkNN. Due to space limitation, we only present the average time it takes when using different number



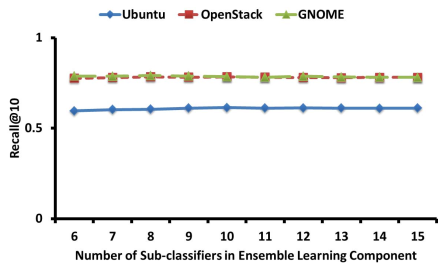


Fig. 5. Recall@5 and recall@10 of *PkgRec* for the 3 datasets when varying the number of sub-classifiers in the ensemble learning component

TABLE VIII
PRECISION@5 OF *PkgRec*, LDA-KL AND MLKNN ON THE 3 DATASETS.

Approaches	Ubuntu	OpenStack	GNOME	Average.
PkgRec	0.200	0.152	0.257	**0.203**
LDA-KL	0.109	0.130	0.151	**0.130**
Improve. LDA-KL	**83.49%**	**16.92%**	**70.20%**	**56.15%**
MLkNN	0.116	0.134	0.128	**0.126**
Improve. MLkNN	**72.41%**	**13.43%**	**100.78%**	**61.11%**

TABLE IX
PRECISION@10 OF *PkgRec*, LDA-KL AND MLKNN ON THE 3 DATASETS.

Approaches	Ubuntu	OpenStack	GNOME	Average.
PkgRec	0.122	0.089	0.136	**0.116**
LDA-KL	0.076	0.080	0.099	**0.085**
Improve. LDA-KL	**60.53%**	**11.25%**	**37.37%**	**36.47%**
MLkNN	0.079	0.080	0.084	**0.081**
Improve. MLkNN	**54.43%**	**11.25%**	**61.90%**	**43.21%**

of folds (from 1 fold to 9 folds) as training data. The results are shown in Table X.

Compared to other approaches, *PkgRec* has the fastest model building time; this is the case since *PkgRec* will build sub-classifiers using the disjoint sets of the training data (which are smaller in size), while the remaining approaches train their models using all the training data. We also notice that the model prediction time of *PkgRec* is longer than baseline approaches. However, we believe it is still acceptable (e.g., the average time to predict packages for Ubuntu bug reports is less than 1 minute).

Threats to Validity. Threats to internal validity relates to errors and bias in our experiments. We have double checked and fully tested our code; still there could be errors that we did not notice. For the ground truth creation, to ensure the affected packages of a bug report are truly affected, we only consider affected packages with the status of "Fix Released" in a bug report. Moreover, our settings for the parameters of *PkgRec* might not be optimal. To minimize this threat we have investigated the performance of *PkgRec* when varying the number of nearest neighbors in MLkNN and the number of sub-classifiers in ensemble learning component. The results show that our approach is relatively stable with different parameter settings.

TABLE X
MODEL BUILDING TIME, AND PREDICTION TIME FOR PKGREC, LDA-KL, AND MLKNN (IN SECONDS).

Project	PkgRec	LDA-KL	MLkNN
Model building time			
Ubuntu	134.85	525.17	154.52
OpenStack	64.89	201.81	103.88
Gnome	11.02	45.96	14.10
Average.	70.25	257.65	90.83
Prediction time			
Ubuntu	31.49	0.86	21.18
OpenStack	14.64	0.18	11.90
Gnome	2.55	0.13	1.89
Average.	16.23	0.39	11.66

Threats to external validity relates to the generalizability of our results. We have analyzed 42,094 bug reports from 3 projects. In the future, we plan to reduce this threat further by analyzing more bug reports from more projects. Another threat is that our approach needs to observe the distribution of testing data. Note that the use of testing data to train a classifier is allowable as long as their class labels are not used [24]. For a large project, many bug reports are often opened for a long time and need to be queued before developers start working on them. For these reports, we know their contents but not the labels (i.e., which packages are affected), and we can use them to tune our model once the number of newly received bug reports are sufficiently many. However, the above may not be true for projects that receive a small number of bug reports. We plan to investigate the performance of our approach using less bug reports as testing data in future work.

Threats to construct validity refers to the suitability of our evaluation measures. We use recall@5 and recall@10 which follows previous software engineering studies [2]–[4]. Thus, we believe there is little threat to construct validity.

VII. RELATED WORK

In this section, we briefly review studies that recommend components affected by a bug report, studies on bug triaging, and other studies on bug report management.

Component Recommendation. The most related work to our paper is conducted by Somasundaram and Murphy [15]. They investigated the performance of different approaches to automatically recommend affected components given a bug report. They found that LDA-KL performs the best. LDA-KL first applies Latent Dirichlet Allocation (LDA) [25] to compute the average topic distribution of a collection of training bug reports belonging to the same unit (i.e., component) and the the topic distribution of a new bug report. Then it computes the Kullback-Leibler (KL) divergence between the topic distribution of the new bug report and that of each component. The components with the least divergence are recommended for the new bug report.

In our work, we focus on recommending software packages that would be affected by a bug report. A component usually refers to internal part of the software project, while a package can be a third-party code. In Somasundaram and Murphy's work, the number of components is relatively small (i.e., it ranges between 13 and 26), while in our datasets, the number of packages ranges between 71 and 341. Our experiment results also show that *PkgRec* outperforms LDA-KL by a wider margin when the number of packages is large (e.g., *PkgRec* achieves an improvement of 76.21% over LDA-KL for recall@5 on Ubuntu dataset, which has 341 packages).

Studies on Bug Triaging. There are a number of approaches that recommend developers to a bug report (aka. automated bug triaging approaches) [5], [14], [26]–[31]. Anvik et al. [26] and Cubranic et al. [27] were the first to investigate this problem, and they used machine learning approaches such as SVM, Naive Bayes, and C4.8 to solve it. Jeong et al. [28] proposed a graph model based on Markov chains, which captures bug tossing history to improve bug triaging prediction accuracy. Bhattacharya et al. [14] furthered improved the accuracy of the approach by Jeong et al. by proposing a multi-feature tossing graph. Naguib et al. [29] proposed a LDA-based approach to compare a bug report with developers in topic space. Tamrawi et al. [5] proposed Bugzie, which uses a fuzzy set and cache-based approach to increase the accuracy of bug triaging.

There are also a number of automatic bug triaging approaches that use other information sources (e.g., source code comments and commit logs) in addition to bug reports to recommend developers to bug reports. A number of approaches [32]–[34] use feature location techniques to find program units (e.g., files or classes) that are related to a change request (i.e., bug report or feature request) and then mine comments in source code files or commits in version control systems to recommend appropriate developers. However, the accuracy of feature location techniques is often still low, which may impact the performance of the approaches above [35]. Also, the quality of the commit logs and source code comments can be poor due to different reasons, such as unavailability of authorship information for authors without commit rights in CVS and SVN repositories [36], outdated comments [37], etc.

In this paper, we focus on recommending software packages that are possibly affected by a bug report. Our work is complementary to bug triaging, since it can also help in finding appropriate developers in different groups (who are responsible for these packages) to collaborate in bug fixing.

Studies on Bug Localization. There are a number of bug localization approaches that locate buggy code in different granularities (e.g., in file, class or method level) for a bug report [38]–[43]. Zhou et al. [38] proposed BugLocator, an information retrieval based approach that ranks all files based on the textual similarity between the initial bug report and the source code using a revised Vector Space Model (rVSM), taking into consideration information about similar bugs that have been fixed before. Nguyen et al. [39] developed a specialized topic model to narrow down the search space of buggy files given a bug report. Ye et al. [40] proposed an adaptive ranking approach that leverages domain knowledge extracted from bug reports and source files such as lexical similarity, code change history and so on. Wen et al. [41] proposed Locus, an IR-based approach to locate bugs using software changes, which provides contextual clues for bug-fixing.

Our work is different from these prior work, since they require the access of source code while most of the packages in our dataset are in the binary level. Also, our approach can work as a complement to bug localization techniques. When a large project contains a large number of packages, it would be time consuming to scan all the source code in every package. The maintainers of the project could first apply our approach to locate the most suspect packages, and then inform the corresponding developers of those packages to further locate the buggy code.

Other Studies on Bug Report Management. There are also many other studies that have been proposed to help developers deal with a large number of bug reports [16], [20], [44]–[56]. Rastkar et al. [44] designed a conversation-based extractive summary generator to produce summaries for bug reports. Zanetti et al. [45] proposed a social network based approach to predict valid bugs in open source projects. Herzig et al. [46] conducted an empirical study on the impact of misclassification on earlier studies of bug prediction and recommended manual data validation for future studies. Zimmermann et al. [47] performed an empirical study on the reopened bugs in the Microsoft Windows operating system. Wang et al. [20] used execution trace information (i.e., list of executed methods) of bug-revealing runs and natural language information contained in bug reports to identify duplicate bug reports. Sun et al. [48] proposed a discriminative model based approach for duplicate bug report detection and they [49] also proposed a retrieval function which extends BM25F, to measure the similarity between two bug reports.

Our work is orthogonal to the above studies; we focus on recommending software packages that are possibly affected by a bug report, which is a different problem.

VIII. Conclusion and Future Work

In this paper, we propose *PkgRec* to automatically recommend packages that are likely to be affected by a bug report. Our approach consists of 2 components: name matching component and ensemble learning component. We evaluate *PkgRec* on 3 datasets with 42,094 bug reports in total. The experiment results show that, *PkgRec* outperforms other state-of-the-art approaches. On average across the 3 datasets, *P-kgRec* improves the recall@5 and recall@10 scores of LDA-KL by 47.25% and 31.41%, and MLkNN by 52.49% and 37.36%, respectively. In future work, we plan to improve the performance of *PkgRec* further. We also plan to experiment with more bug reports from more projects.

Acknowledgment. This work was partially supported by NSFC Program (No. 61602403 and 61572426), and National Key Technology R&D Program of the Ministry of Science and Technology of China (No. 2015BAH17F01).

References

[1] D. Bertram, A. Voida, S. Greenberg, and R. Walker, "Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams," in *Proceedings of the 2010 ACM conference on Computer supported cooperative work*. ACM, 2010, pp. 291–300.

[2] X. Xia, D. Lo, X. Wang, and B. Zhou, "Tag recommendation in software information sites," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 287–296.

[3] W. Wu, W. Zhang, Y. Yang, and Q. Wang, "Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking," in *2011 18th Asia-Pacific Software Engineering Conference*. IEEE, 2011, pp. 389–396.

[4] X. Xie, W. Zhang, Y. Yang, and Q. Wang, "Dretom: Developer recommendation based on topic models for bug resolution," in *Proceedings of the 8th international conference on predictive models in software engineering*. ACM, 2012, pp. 19–28.

[5] 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.

[6] 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.

[7] G. Salton, A. Wong, and C.-S. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[8] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information processing & management*, vol. 24, no. 5, pp. 513–523, 1988.

[9] G. Salton and M. J. McGill, "Introduction to modern information retrieval," 1986.

[10] J. Han, J. Pei, and M. Kamber, *Data mining: concepts and techniques*. Elsevier, 2011.

[11] T. G. Dietterich, "Ensemble methods in machine learning," in *International workshop on multiple classifier systems*. Springer, 2000, pp. 1–15.

[12] M. Woźniak, M. Graña, and E. Corchado, "A survey of multiple classifier systems as hybrid systems," *Information Fusion*, vol. 16, pp. 3–17, 2014.

[13] J. M. Al-Kofahi, A. Tamrawi, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Fuzzy set approach for automatic tagging in evolving software," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–10.

[14] 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.

[15] K. Somasundaram and G. C. Murphy, "Automatic categorization of bug reports using latent dirichlet allocation," in *Proceedings of the 5th India software engineering conference*. ACM, 2012, pp. 125–130.

[16] X. Xia, L. Bao, D. Lo, and S. Li, "automated debugging considered harmful considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 2016, pp. 267–278.

[17] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 165–176.

[18] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 international symposium on software testing and analysis*. ACM, 2011, pp. 199–209.

[19] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 499–510.

[20] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 461–470.

[21] F. Wilcoxon, *Individual Comparisons by Ranking Methods*. Springer New York, 1992.

[22] H. Abdi, "The bonferonni and šidák corrections for multiple comparisons," *Encyclopedia of measurement and statistics*, vol. 3, pp. 103–107, 2007.

[23] N. Cliff, *Ordinal methods for behavioral data analysis*. Lawrence Erlbaum Associates, 1996.

[24] 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.

[25] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[26] 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.

[27] D. Čubranić, "Automatic bug triage using text categorization," in *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer, 2004.

[28] 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.

[29] H. Naguib, N. Narayan, B. Brügge, and D. Helal, "Bug report assignee recommendation using activity profiles," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE, 2013, pp. 22–30.

[30] X. Xia, D. Lo, Y. Ding, J. M. Al-Kofahi, T. N. Nguyen, and X. Wang, "Improving automated bug triaging with specialized topic model," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 272–297, 2017.

[31] X. Xia, D. Lo, X. Wang, and B. Zhou, "Dual analysis for recommending developers to resolve bugs," *Journal of Software: Evolution and Process*, vol. 27, no. 3, pp. 195–220, 2015.

[32] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Hammad, "Assigning change requests to software developers," *Journal of Software: Evolution and Process*, vol. 24, no. 1, pp. 3–33, 2012.

[33] M. Linares-Vásquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk, "Triaging incoming change requests: Bug or commit history, or code authorship?" in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 451–460.

[34] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani, "Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 2–11.

[35] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 345–355.

[36] C. Kolassa, D. Riehle, and M. A. Salim, "A model of the commit size distribution of open source," in *International Conference on Current Trends in Theory and Practice of Computer Science*. Springer, 2013, pp. 52–66.

[37] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan, "On the relationship between comment update practices and software bugs," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2293–2304, 2012.

[38] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 14–24.

[39] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, 2011, pp. 263–272.

[40] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *ACM Sigsoft International Symposium on Foundations of Software Engineering*, 2014, pp. 689–699.

[41] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 2016, pp. 262–273.

[42] X. Xia, D. Lo, X. Wang, C. Zhang, and X. Wang, "Cross-language bug localization," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 275–278.

[43] Y. Zhang, D. Lo, X. Xia, T.-D. B. Le, G. Scanniello, and J. Sun, "Inferring links between concerns and methods with multi-abstraction vector space model," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 2016, pp. 110–121.

[44] S. Rastkar, G. C. Murphy, and G. Murray, "Automatic summarization of bug reports," *IEEE Transactions on Software Engineering*, vol. 40, no. 4, pp. 366–380, 2014.

[45] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer, "Categorizing bugs with social networks: a case study on four open source software communities," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 1032–1041.

[46] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 392–401.

[47] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy, "Characterizing and predicting which bugs get reopened," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1074–1083.

[48] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 45–54.

[49] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 253–262.

[50] X. Xia, D. Lo, E. Shihab, X. Wang, and B. Zhou, "Automatic, high accuracy prediction of reopened bugs," *Automated Software Engineering*, vol. 22, no. 1, pp. 75–109, 2015.

[51] X.-L. Yang, D. Lo, X. Xia, Q. Huang, and J.-L. Sun, "High-impact bug report identification with imbalanced learning strategies," *Journal of Computer Science and Technology*, vol. 32, no. 1, pp. 181–198, 2017.

[52] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun, "Combining word embedding with information retrieval to recommend similar bug reports," in *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*. IEEE, 2016, pp. 127–137.

[53] Z. Wan, D. Lo, X. Xia, and L. Cai, "Bug characteristics in blockchain systems: a large-scale empirical study," in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 413–424.

[54] T. Zhang, J. Chen, H. Jiang, X. Luo, and X. Xia, "Bug report enrichment with application of automated fixer recommendation," in *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 2017, pp. 230–240.

[55] X. Xia, D. Lo, X. Wang, and B. Zhou, "Automatic defect categorization based on fault triggering conditions," in *Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on*. IEEE, 2014, pp. 39–48.

[56] Y. Tian, D. Lo, X. Xia, and C. Sun, "Automated prediction of bug report priority using multi-factor analysis," *Empirical Software Engineering*, vol. 20, no. 5, pp. 1354–1383, 2015.