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 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: Ubuntu², OpenStack³ and GNOME⁴. In total, we analyze 42,094 bug reports.

1In 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.
2https://bugs.launchpad.net/ubuntu
3https://bugs.launchpad.net/openstack
4https://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.

**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/

<table>
<thead>
<tr>
<th>Bug ID: #804435</th>
<th>Summary: Wallpaper is loaded twice with different alignment by gnome-session and nautilus (Oneiric)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Affects:</strong></td>
<td>nautilus (Status: Invalid) GTK+ (Status: Fix Released) unity-2d (Status: Fix Released)</td>
</tr>
<tr>
<td><strong>Bug Description:</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>ProblemType:</strong></td>
<td>Bug</td>
</tr>
<tr>
<td><strong>DistroRelease:</strong></td>
<td>Ubuntu 11.10</td>
</tr>
<tr>
<td><strong>Package:</strong></td>
<td>nautilus 1:3.1.2-0ubuntu2</td>
</tr>
<tr>
<td><strong>ProcVersionSignature:</strong></td>
<td>Ubuntu 3.0.2-3-generics 3.0.0-rc4</td>
</tr>
<tr>
<td><strong>Uname:</strong></td>
<td>Linux 3.0.2-generic i686</td>
</tr>
<tr>
<td><strong>Architecture:</strong></td>
<td>i386</td>
</tr>
<tr>
<td><strong>Date:</strong></td>
<td>Fri Jul 1 18:51:10 2011</td>
</tr>
<tr>
<td><strong>ProcEnviron:</strong></td>
<td>PATH=(custom, no user) LANG=it_IT.UTF-8 SHELL=/bin/bash</td>
</tr>
<tr>
<td><strong>SourcePackage:</strong></td>
<td>nautilus</td>
</tr>
<tr>
<td><strong>UnameUpgradeStatus:</strong></td>
<td>No upgrade log present (probably fresh install)</td>
</tr>
</tbody>
</table>
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) \( X_{\text{new}} \), 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 \( X_{\text{new}} \), MLkNN first finds its k-nearest neighbors \( knn(X_{\text{new}}) \) from the training dataset. Then for each label (i.e., package) \( l \) in the label set \( L \), it counts the number of instances in \( knn(X_{\text{new}}) \) that are assigned to label \( l \), denoted as \( C_{X_{\text{new}}}(l) \).

**MLkNN Confidence Score.** With the membership counting score \( C_{X_{\text{new}}}(l) \) for each label \( l \), we consider two events: \( H_1^l \) is the event that \( X_{\text{new}} \) is assigned to \( l \), and \( H_0^l \) is the event that \( X_{\text{new}} \) 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 \( knn(X_{\text{new}}) \). Then, the MLkNN confidence score for \( l \) and \( X_{\text{new}} \) is the probability that the \( X_{\text{new}} \) is assigned to \( l \), given exactly \( C_{X_{\text{new}}}(l) \) instances in \( knn(X_{\text{new}}) \) are assigned to label \( l \). Formally, we have the following equation:

\[
ML(X_{\text{new}}, l) = P(H_1^l | E_{C_{X_{\text{new}}}}^l) = \frac{P(\mathit{H}_1^l \times P(E_{C_{X_{\text{new}}}}^l | \mathit{H}_1^l)}{\sum_{i \in \{0,1\}} P(H_1^l) \times P(E_{C_{X_{\text{new}}}}^l | \mathit{H}_1^l)}
\]

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

\[
ML(X_{\text{new}}, H_1^l) = \frac{P(H_1^l) \times P(E_{C_{X_{\text{new}}}}^l | H_1^l)}{P(H_1^l)} \times P(E_{C_{X_{\text{new}}}}^l | H_1^l)
\]

The parameters of \( P(H_1^l), P(H_0^l), P(E_m^l | H_1^l) \) and \( P(E_m^l | H_1^l) \) 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 \( X_{\text{new}} \). In this paper, instead of outputting predicted labels for \( X_{\text{new}} \), we modify MLkNN to recommend the top-k labels that have the highest confidence scores, denoted as \( MLkNN(X_{\text{new}}, H_1^l) \).

### 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:

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

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 \( \text{wordsInName}(p) \). Then we count the number of single words that appear in the title or description of a bug report, denoted as \( \text{nameHits}(b, p) \). 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 $\alpha_i$ denotes the weight factor of the $i^{th}$ sub-classifier $C_i$. The value of $\alpha_i$ is given by the following equation:

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

In the above equation, $EC(C_i, \text{MockSet})$ 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 $\text{Ensemble}(b, p)$, which is given by the following equation:

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

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 $\alpha_i$ (Line 16-19). Finally, we return $\vec{\alpha}$ (Line 20).

### C. PkgRec: A Composite Approach

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

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

Where $\gamma_1, \gamma_2 \in [0, 1]$ represent the weight factors of name matching score and ensemble learning score to the overall $\text{PkgRec}$ score. Similar to Algorithm 1, the values of $\gamma_1$ and $\gamma_2$ 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, $\text{PkgRec}$ 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 $\text{PkgRec}$ 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
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:

\[
Recall@k = \frac{1}{m} \sum \frac{|P_i \cap D_i|}{|D_i|}
\]

\[
Precision@k = \frac{1}{m} \sum \frac{|P_i \cap D_i|}{|P_i|}
\]

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 ($\delta$) [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 $\delta = -1$ 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 $\delta = 0$ indicates the two approaches are completely overlapping. Table III describes the meaning of different Cliff’s delta values and their corresponding interpretation [23].

### TABLE II

**Statistics of collected bug reports.**

<table>
<thead>
<tr>
<th>Project</th>
<th>Time</th>
<th># Reports</th>
<th># Features</th>
<th># Packages</th>
<th># Avg. Affects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ubuntu</td>
<td>2004-12-26 - 2011-11-29</td>
<td>18,530</td>
<td>3,808</td>
<td>341</td>
<td>2,005</td>
</tr>
<tr>
<td>Openstack</td>
<td>2012-05-16 - 2014-10-09</td>
<td>18,207</td>
<td>2,839</td>
<td>71</td>
<td>1,203</td>
</tr>
<tr>
<td>GNOME</td>
<td>2005-06-18 - 2016-05-26</td>
<td>5,357</td>
<td>3,016</td>
<td>167</td>
<td>1,757</td>
</tr>
</tbody>
</table>

### TABLE III

**Cliff’s Delta and the Effectiveness Level**

| Cliff’s Delta ($|\delta|$) | Effectiveness Level |
|----------------|---------------------|
| $|\delta| < 0.147         | Negligible          |
| $0.147 \leq |\delta| < 0.33$         | Small               |
| $0.33 \leq |\delta| < 0.474$         | Medium              |
| $|\delta| \geq 0.474$     | Large               |

\[^6^\] http://mulan.sourceforge.net/
\[^7^\] http://gibblda.sourceforge.net/
\[^8^\] http://github.com/tkdsheep/MultiPackage
Results. Table IV compares recall@5 and recall@10 of PkgRec and LDA-KL. Table V compares recall@5 and recall@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. From 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 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 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

<table>
<thead>
<tr>
<th>Projects</th>
<th>Recall@5</th>
<th>Recall@10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ubuntu</td>
<td>0.511</td>
<td>0.334</td>
</tr>
<tr>
<td>OpenStack</td>
<td>0.679</td>
<td>0.600</td>
</tr>
<tr>
<td>GNOME</td>
<td>0.737</td>
<td>0.398</td>
</tr>
<tr>
<td>Average</td>
<td>0.642</td>
<td>0.444</td>
</tr>
</tbody>
</table>

TABLE VII

<table>
<thead>
<tr>
<th>Projects</th>
<th>Recall@5</th>
<th>Recall@10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ubuntu</td>
<td>0.511</td>
<td>0.361</td>
</tr>
<tr>
<td>OpenStack</td>
<td>0.679</td>
<td>0.232</td>
</tr>
<tr>
<td>GNOME</td>
<td>0.737</td>
<td>0.659</td>
</tr>
<tr>
<td>Average</td>
<td>0.642</td>
<td>0.417</td>
</tr>
</tbody>
</table>

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.

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
Results. Figure 4 presents the recall@5 and recall@10 scores of PkgRec on 3 datasets when varying the number of nearest neighbor 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 MLkNN 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
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". 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.

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 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 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 allows our approach to be more generalizable. 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 allows our approach to be more generalizable. 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.**

**VI. 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 then 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
PkgRec achieves an improvement of 76.21% over LDA-KL with 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). There are 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 use 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 use fuzzy set and cache-based approach to increase the accuracy of bug triaging.

Our work is orthogonal to the above studies; we focus on finding appropriate developers in different groups (who are responsible for these packages) to collaborate in bug fixing.

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. 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, PkgRec 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).