Reducing Bug Triaging Confusion by Learning from Mistakes with a Bug Tossing Knowledge Graph

Yanqi Su
Australian National University
Australia
Yanqi.Su@anu.edu.au

Zhenchang Xing*
Australian National University
Australia
Zhenchang.Xing@anu.edu.au

Xin Peng
Fudan University
China
pengxin@fudan.edu.cn

Xin Xia
Monash University
Australia
xin.xia@monash.edu

Chong Wang
Fudan University
China
wangchong20@fudan.edu.cn

Xiwei Xu
Data61, CSIRO
Australia
Xiwei.Xu@data61.csiro.au

Liming Zhu†
Data61, CSIRO
Australia
Liming.Zhu@data61.csiro.au

***Abstract*—Assigning bugs to the right components is the prerequisite to get the bugs analyzed and fixed. Classification-based techniques have been used in practice for assisting bug component assignments, for example, the BugBug tool developed by Mozilla. However, our study on 124,477 bugs in Mozilla products reveals that erroneous bug component assignments occur frequently and widely. Most errors are repeated errors and some errors are even misled by the BugBug tool. Our study reveals that complex component designs and misleading component names and bug report keywords confuse bug component assignment not only for bug reporters but also developers and even bug triaging tools. In this work, we propose a learning to rank framework that learns to assign components to bugs from correct, erroneous and irrelevant bug-component assignments in the history. To inform the learning, we construct a bug tossing knowledge graph which incorporates not only goal-oriented component tossing relationships but also rich information about component tossing community, component descriptions, and historical closed and tossed bugs, from which three categories and seven types of features for bug, component and bug-component relation can be derived. We evaluate our approach on a dataset of 98,587 closed bugs (including 29,100 tossed bugs) of 186 components in six Mozilla products. Our results show that our approach significantly improves bug component assignments for both tossed and non-tossed bugs over the BugBug tool and the BugBug tool enhanced with component tossing relationships, with >20% Top-k accuracies and >30% NDCG@k (k=1,3,5,10).**

***Index Terms*—Bug Triaging, Learning to Rank, Knowledge Graph**

I. Introduction

Large software projects (e.g., Mozilla) receive a huge number of bug reports every day. To get the attention of suitable developers on the bugs quickly, it is essential to assign bugs to the right product and component (referred to as bug triaging in this work) whose developers have the expertise to review, analyze and fix the bugs. Manual bug triaging is a labor-intensive task, and is often error prone [1]–[4]. If the bug is not assigned to the right component, it has to be re-assigned (or tossed) to the correct one. For example, the Mozilla's Bug 1194529 was initially assigned to Toolkit::Password Manager (Product::Component) and then was tossed to Firefox::about:logins and fixed there. In this work, we refer to bugs that have been tossed as tossed bugs, and bugs that are initially assigned to correct components as non-tossed bugs. We refer to the component where a bug gets fixed as resolver, and the component(s) that a bug is erroneously assigned to as bystander(s). For example, Bug 1194529 is a tossed bug, and its resolver is Firefox::about:logins. This bug has a bystander Toolkit::Password Manager.

Since the seminal work by Anvik et al [5], machine learning techniques have been used to assist bug triaging [1], [2]. These techniques treat bug triaging as a multi-class classification task, taking as input the information of a bug (e.g., bug summary) and predicting the most likely resolving component as the class label. BugBug [6] is such a tool developed by the Mozilla product team. We investigate 124,477 bugs of 186 components of the six Mozilla products. We find that 38,374 (30.8%) bugs have been tossed at least once. Overall, tossed bugs take 2.9 times longer to get fixed than non-tossed bugs. Compared with the statistics of bug tossing phenomena reported in early studies [1], [2], [4], the bug tossing situation actually does not change much, even after a long-time deployment of machine learning based bug triaging techniques.

As detailed in Section II, bug tossing is mostly caused by repeated erroneous bug-component assignments. These repeated errors (no matter manually or by machine learning) stem from the lack of effective mechanisms to model and differentiate confusing concepts related to bugs and software components, in particular complex component designs and relationships and misleading component names and bug report keywords. In practice, developers often resort to additional information such as detailed component descriptions and component communities, rather than relying on only the information in bug reports, to correct erroneous bug-component assignments. Unfortunately, current bug triaging techniques [1], [2], [6] do not make use of such additional information. Furthermore,

*Corresponding author.
†Also with University of New South Wales.

the formulation of bug triaging as a multi-class classification problem learns from each bug-component assignment independently. However, this is not sufficient to distinguish misleading information from key problem information in bug reports and how different information relates to different components.

In fact, bug tossing history archives rick knowledge about bug-component relations including both correct and erroneous assignments. We propose a novel framework (called LR-BKG, short for Learn-to Rank with Bug tossing Knowledge Graph) to reduce bug triaging confusion by learning from bug tossing history. Inspired by the goal-oriented developer tossing graph [7], our approach builds a goal-oriented component tossing graph from bug tossing history. Driven by our empirical observation, we enrich this basic tossing graph into a bug tossing knowledge graph by attaching three types of information on each component (component name/description, non-tossed bugs and tossed bugs), and by detecting component communities based on historical tossing paths. We develop three categories of features: bug features, component features and bug-component relation features. The feature design makes full use of the concept- and community-enriched bug tossing graph. Instead of traditional multi-class classification, we adopt a learning-to-rank framework which learns to differentiate confusing bug-component relationships by contrasting correct and erroneous bug assignments.

To evaluate our approach, we collect 98,587 closed bugs (including 29,100 tossed bugs) of 186 components in six Mozilla products. To simulate real-world context, we sort these bugs by their creation time and split them at 25th February, 2020. This gives us 80% of bugs as "historical" training data and the rest 20% as "future" bugs to test the trained model. We compare our approach with two baselines: BugBug and BugBug with tossing graph. Overall, LR-BKG achieves 20% or higher accuracy in recommending resolver components than the two baselines at all Top-ks (k=1, 3, 5, 10). It achieves same-level improvement for both tossed and non-tossed bugs, and achieves improvement for 73%-85% of components at different Top-ks. Furthermore, LR-BKG can better rank bystander components that have historical tossing relationships with resolver components (30% or higher NDCG@k (k=1, 3, 5, 10) than BugBug). Our feature importance analysis shows that all three categories of features as a whole contribute to the significant improvement LR-BKG achieves.

This paper makes the following contributions:

- We conduct an empirical study on the root cause of repeated bug tossing, which sheds the light on novel ways to reduce bug triaging confusion.
- We propose a learning-to-rank framework that learns to distinguish correct, erroneous and irrelevant bug-component assignments, based on a rich set of features derived from our novel bug tossing knowledge graph.
- Our experiments confirm our approach's superior performance than the tool used in development practice, and confirm the effectiveness of our novel feature design. Our replication package can be found here[1].

TABLE I: Tossed-In Bug Percentage of Product::Component

Tossed-In (%)	2.9-20%	20-40%	40-60%	60-80%	80-84.9%
#P::C	39	96	39	11	1

II. Empirical Study

We conduct an empirical study of bug tossing phenomena to answer the following three research questions:

RQ1: Does bug tossing occur frequently and widely?
RQ2: Is bug tossing accidental mistake or repeated error?
RQ3: What causes such repeated bug tossing?

A. Dataset

We use software products of Mozilla Foundation as study subjects. We crawl 124,477 bugs from the Mozilla's Bugzilla website (https://bugzilla.mozilla.org/home) which involve six products (Firefox, Firefox Build System, Toolkit, Core, DevTools, WebExtensions) and their 186 components. Although Mozilla's Bugzilla website has thousands of components, many components have very few bugs. We restrict our study to the components with at least 1% of the number of bugs of the component with the largest number of bugs. These components cover very frontend and backend features of Mozilla products. Thanks Bugzilla, we can extract the complete bug assignment history of each bug. The history contains initially assigned component and all subsequent components a bug has been tossed to in chronological order.

B. RQ1: Bug Tossing Frequency and Impact

Among 124,477 bugs, 30.8% (i.e., 38,374 bugs) has been tossed. Table I shows the tossed-in bug percentage of a component (i.e., the bugs that are currently assigned to a component but were not initially assigned to this component). None of 186 components have zero tossed-in bug percentage. That is, all 186 components have some bugs that has been tossed-in from other components. Toolkit::Blocklist Policy Requests is the only component whose tossed-in bug percentage is below 5% (in particular 18/615 bugs, 2.9%). Tossed-in bug percentage is 20-40% for 96 components and 40-60% for 39 components. For 12 components, tossed-in bug percentage is even higher than 60%. For example, out of 258 bugs of Firefox Build System::Android Studio and Gradle Integration, 219 are tossed-in bugs which give tossed-in bug percentage 84.9%. For the tossed bugs, the time it takes to close them is on average about 2.9 times longer than the non-tossed bugs.

> Bug tossing not only occurs frequently but also widely on Mozilla product components. In fact, bug tossing statistics do not change much in the past 15 years [1], [2], [4].

C. RQ2: Accidental Mistake or Repeated Error

For each tossed bug, we obtain a path from its initial component to its current component. We identify dis-

[1] https://github.com/SuYanqi/LR-BKG

TABLE II: Examples of Real Tossing Paths

Product::Component	Tossing Path	Frequency
Toolkit:: Password Manager	a. Firefox::Security → Toolkit::Password Manager	6
	b. Toolkit::Password Manager:Site Compatibility → Toolkit::Password Manager	6
	c. Firefox::about:logins → Toolkit::Password Manager	4
Toolkit:: Password Manager: Site Compatibility	a. Firefox::Untriaged → Toolkit::Password Manager → Toolkit::Password Manager:Site Compatibility	23
	b. Toolkit::Password Manager → Toolkit::Password Manager:Site Compatibility	10
Firefox:: about:logins	a. Toolkit::Password Manager → Firefox::about:logins	269
	b. Firefox::Untriaged → Toolkit::Password Manager→ Firefox::about:logins	17
Firefox::Security	a. Core::Security:PSM → Firefox::Security	3
	b. Core::Networking:DNS → Firefox::Security	3
	c. Firefox::Theme →Firefox::Security	2
Core::Security: PSM	a. Firefox::Untriaged → Core::Security:PSM	207
	b. Firefox::Security → Core::Security:PSM	16
	c. Core::Networking → Core::Security:PSM	11
Toolkit::Themes	a. Firefox::Theme → Toolkit::Themes	18
	b. Firefox::Preferences → Toolkit::Themes	6
	c. Toolkit::XUL Widgets → Toolkit::Themes	5
	d. Toolkit::Password Manager → Toolkit::Themes	3
	e. Firefox::Bookmarks & History → Firefox::Theme → Toolkit::Themes	2
Firefox::Theme	a. Firefox::General → Firefox::Theme	33
	b. Firefox::Bookmarks & History → Firefox::Theme	8
	c. Toolkit::Themes → Firefox::Theme	6

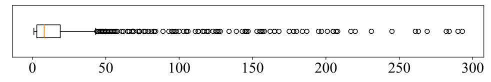
tinct tossing paths and count their occurrence frequencies among all tossed bugs. Table II shows some tossing paths. For example, one tossing path is Firefox::Security → Toolkit::Password Manager, with the frequency 6. That is, 6 bugs have been initially assigned to Firefox::Security, and then tossed to Toolkit::Password Manager, and is currently with Toolkit::Password Manager. This path is just one of the 60 paths along which a bug has been tossed to Toolkit::Password Manager.

We identify 8,487 tossing paths for 38,374 bugs. Only 932 tossing paths occur once, accounting for 11.0% of all tossing paths and 2.4% of all tossed bugs. The rest 7,555 tossing paths have two or more bugs. Fig. 1(a) shows the distribution of tossing path frequencies. We remove 10 large outliers (≥ 307) to show this distribution more clearly. The median frequency is 8, with 3 at 25% quantile and 19 at 75% quantile. 26 tossing paths occurred even more than 200 times. The first tossing path for Firefox::about:logins in Table II is one of these most frequent paths, which occurs 269 times. Fig. 1(b) show the distribution of the number of tossing paths per component, i.e., the number of paths ending with a particular component. The median is 33, with 18 at 25% quantile and 60 at 75% quantile. There are 18 components (e.g. Core::Security:PSM) with 100 or more different tossing paths.

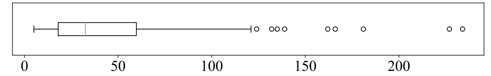
> Accidental mistake may account for only a small percentage of bug tossing, while the majority of bug tossing are caused by repeated erroneous bug-component assignments, and those repeated errors can occur in many different ways.

D. RQ3: Cause of Repeated Bug Tossing

Through observing repeated bug tossing paths, we find that confusing concepts are an important cause of erroneous bug-component assignments. Confusing concepts come from two



(a) Distribution of Tossing Path Frequencies



(b) Distribution of the Number of Tossing Paths per Component

Fig. 1: Distribution of Tossing Paths

sources: complex component designs and relationships, and misleading component names and bug report keywords.

1) Complex Component Designs and Relationships: A complex feature often involves a set of correlated components. For example, according to the Mozilla wiki and a meeting note, password management involves four components: Toolkit::Password Manager for remembering usernames and passwords on sites and filling them when the user returns to the sites, Firefox::about:logins for issues with the Firefox Lockwise Logins and Passwords page, Toolkit::Password Manager:Site Compatibility for issues of autofill, autocomplete or saving of logins/passwords/usernames not working on a specific site, and Firefox::Migration for profile migration from other browsers. In total, 353 bugs that were initially assigned to one of these four components were tossed to the other of these four components. Fig. 3 shows a partial tossing graph constructed from bug tossing paths. Toolkit::Password Manager, Firefox::about:logins and Toolkit::Password Manager:Site Compatibility actually form a bug tossing community (blue nodes and edges). This bug tossing community largely matches the component design for password management.

In face of complex correlated components, bug reporters, especially those without expertise background, often have difficulties in assigning bugs to the correct components. For this problem, Mozilla sets general placeholder components for reporters to report bugs, such as Firefox::General, Firefox::Untriaged. Then, developers have to toss them into right components. Table II shows some examples of tossing paths starting from Firefox::Untriaged or Firefox::General. First, this general-then-toss practice incurs significant burden on developers due to the sheer amount of bugs initially reported to general placeholder components. Second, even developers may not make right tossing decisions in many cases. 4,251 bugs that have been tossed from general components were tossed again (e.g. Firefox::Untriaged → Toolkit::Password Manager → Toolkit::Password Manager:Site Compatibility, Firefox::Untriaged → Toolkit::Password Manager → Firefox::about:logins). Sometimes developers may just randomly assign a bug to one of correlated components and hope that other developers who have the expertise will toss the bug to the right component (Bug 1650593 is such an example).

2) Misleading Names and Keywords: Many components have confusing names, which make them seem relevant to some bugs. Such confusing names often confuse bug reporters and developers. For example, three bug reporters assigned bugs to Toolkit::Form Autofill, but these bugs were tossed

to Toolkit::Password Manager, even though Toolkit::Form Autofill and Toolkit::Password Manager have no relation according to component design. An experienced developer commented one of these bugs (Bug 1595114) "The Form Autofill component is for address and credit card autofill, not logins (see the description)." Moreover, a developer assigned two bugs from Firefox::Untriaged to Toolkit::Form Autofill. Then, another developer tossed these two bugs to Toolkit::Password Manager and commented Bug 1596805 "please don't move login/password bugs to Form Autofill as it is only about address and credit card autofill. Please use Toolkit::Password Manager." The developer who made the mistakes replied "Now I know, but the name of this component is obviously not intuitive." Not only was this developer confused, but BugBug could also make similar mistakes. For example, the user who reports Bug 1653547 states "Bugbug thinks this bug belongs to Form Autofill, please revert in case of error", and it was indeed an error and tossed to Toolkit::Password Manager.

In fact, the information provided in the comments about bug tossing is available in the detailed description of Toolkit::Form Autofill. Unfortunately, many bug reporters, even developers in Mozilla, have no knowledge of these detailed component descriptions. Similarly, 18 Toolkit::Themes bugs were initially assigned to Firefox::Theme. No matter for bug reporters or developers, the two names are confusing. However, the detailed component descriptions tell the differences. Firefox::Theme is responsible for general user interface, user experience, and visual design for the default theme used in Firefox. Bugs about packaged browser themes and about WebExtensions that use the "themes" API should be triaged into Toolkit::Themes.

Bug reports very likely contain misleading information, which may confuse bug reporters, developers, and automatic bug triaging tool BugBug developed by Mozilla. For example, Bug 1194529 states "Ask the user for their OS account password/biometrics before showing the passwords in the password manager." It contains keywords "password" and "password manager", and was initially assigned to Toolkit::Password Manager. But this bug is essentially about an issue in Logins and Passwords page, and thus was tossed to Firefox::about:logins. For Bug 1584846, the situation is the opposite. Its summary states "Separate preference setting for Autofill logins/passwords from Ask to save logins/passwords". It mentions "logins" and "password" a couple of times. This bug was initially assigned to Firefox::about:logins, but was tossed to Toolkit::Password Manager as it is about password autofill that Toolkit::Password Manager is responsible for.

As bug report summary and description is the most important input to machine learning based bug triaging techniques, misleading information in bug reports could also mislead machine learning techniques. For example, misled by the BugBug's recommendation, the bug reporter assigned Bug 1644112 "Password displayed too short with Standard Font 14 in Firefox access data" to Core::Graphics:Text. Unfortunately, this was a completely non-sense assignment. The Bug 1644112 was tossed to Firefox::WebPayments UI, and then Firefox::about:logins and closed there. This mistake by BugBug is because the Bug 1644112's summary contains keywords like "font", "display", "data" that are often in the bug reports correctly assigned to Core::Graphics:Text. However, the key problem of this bug is about password display that Firefox::about:logins is responsible for. The initial non-sense assignment challenges the Core::Graphics:Text developers as they lack experience in handling bugs like Bug 1644112. As a result, they tossed the bug to Firefox::WebPayments UI which was still incorrect, and it took one day before this bug was tossed to the right component Firefox::about:logins.

> Complex component designs and misleading component names and bug report keywords cause a lot of confusion in bug triaging and tossing. Developers often resort to additional information such as detailed component descriptions and component communities to correct erroneous bug assignments. Furthermore, existing bug triaging techniques learn from each bug-component assignment independently, which often cannot effectively distinguish misleading information and key problem information in bug reports.

III. APPROACH

Inspired by our empirical study findings, we propose a novel bug triaging approach with two aims: improve bug triaging accuracy and avoid tossing-irrelevant bug triaging. As shown in Figure 2, our approach builds on a novel bug triaging knowledge graph, which enriches basic goal-oriented component tossing graph with rich information about component, component community, and correctly and erroneously assigned bugs. Instead of only bug-report centric features, we derive a rich set of features from bug tossing knowledge graph, which represent additional knowledge (i.e., component descriptions and component community) commonly used to correct erroneous bug assignments, and rich relations between an input bug and those correctly or erroneously assigned bugs. In contrast to learning from each bug-component assignment independently, our approach adopts a learning-to-rank framework which ranks bug-component relevance by contrasting correct, erroneous and irrelevant bug assignments in the history.

A. Construction of Bug Tossing Knowledge Graph

To make effective use of historical bug tossing information, we first construct a bug tossing knowledge graph from which a rich set of bug and component features can be derived.

1) Goal-Oriented Bug Tossing Graph: Bug tossing relationships among components provide useful information for learning the relevance of a bug to its resolver component as opposed to all other bystander components that the bug has been erroneously assigned to. These relationships are captured in real bug tossing paths (see Table II for some examples). We convert real tossing paths into goal-oriented tossing paths. The conversion is straightforward. For each component (except the resolver component) in a real tossing path, we create a goal-oriented tossing path from this component to the resolver component. Consider the real tossing path Firefox::Bookmarks

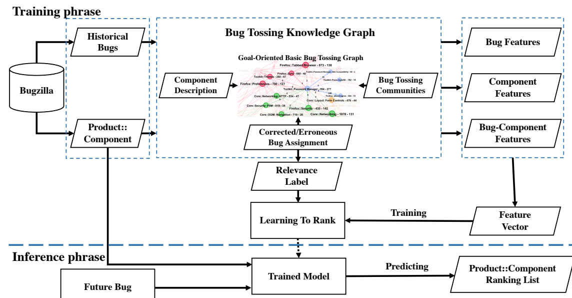


Fig. 2: Approach Overview

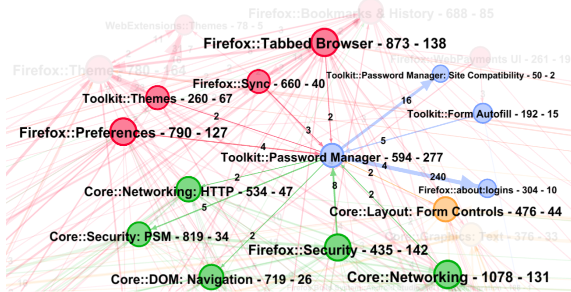


Fig. 3: Goal-Oriented Bug Tossing Knowledge Graph

& History → Firefox::Theme → Toolkit::Themes in Table II. From this real tossing path, we create two goal-oriented tossing paths: Firefox::Bookmarks & History → Toolkit::Themes and Firefox::Theme → Toolkit::Themes. After processing all real tossing paths, we identify distinct goal-oriented tossing paths and count their occurrence frequencies

Goal-oriented tossing paths can avoid the two limitations of real tossing paths. First, real tossing paths capture only what have occurred in the past, but they will not cover all possible tossing paths. For example, the presence of the tossing path Firefox::Bookmarks & History → Firefox::Theme → Toolkit::Themes indicates that Firefox::Bookmarks & History → Toolkit::Themes is possible to occur. In fact, after 2.5 years that Firefox::Bookmarks & History → Firefox::Theme → Toolkit::Themes occurred for Bug 1468080 and Bug 1476790, Firefox::Bookmarks & History → Toolkit::Themes occurred for Bug 1671000. However, before Bug 1671000 appeared, Firefox::Bookmarks & History → Toolkit::Themes is not a real tossing path for 2.5 years.

Second, the bystander components (e.g., Firefox::Theme in the above path) may be somehow relevant to the tossed bug, but they are definitely not as relevant as the resolver component (e.g., Toolkit::Themes in the above path). Even worse, in many cases bystander components may not be relevant at all to the tossed bug. For example, in the tossing path Core::Graphics:Text → Firefox::WebPayments UI → Firefox::about:logins for Bug 1644112, Firefox::WebPayments UI is irrelevant as the Core::Graphics:Text developers did not know where is appropriate to toss Bug 1644112 either. Learning from such tossing mistakes would not be beneficial.

Given all goal-oriented tossing paths obtained, we construct a goal-oriented bug tossing graph. The graph nodes are components. The edges are directed and represent goal-oriented tossing paths from one component to another. Fig. 3 shows a subgraph of the goal-oriented bug tossing graph constructed in our empirical study on Mozilla product::components. The nodes of Toolkit::Password Manager and its direct neighbors are highlighted. From the faded background, we can observe rather complex tossing relationships among Mozilla product::components. For each component node, the graph records two metrics: the number of bugs currently assigned (including initially assigned and tossed from other components) to this component (e.g., 594 for Toolkit::Password Manager); the number of bugs tossed from this component to other components (e.g., 277 for Toolkit::Password Manager). For each edge, the graph records one metric: the occurrence frequencies of the tossing between two components (e.g., 240 from Toolkit::Password Manager to Firefox::about:logins).

2) Bug Tossing Community Detection: Frequent tossing relationships among components will form some bug tossing communities in the bug tossing graph. In this work, we use the community detection algorithm [8] to detect bug tossing communities in the bug tossing graph. This algorithm is based on modularity optimization. We use the implementation in the Gephi tool. For community detection, we filter out edges with frequency=1 as we deem them as likely accidental.

In the tossing graph used in our empirical study, the algorithm detects 18 communities covering 186 components. The community size is 10.3 ± 16.2. In Fig. 3, nodes and edges with the same color belong to one community detected by the algorithm. Red and green communities are two large communities which involve 48 and 46 components, respectively. As we highlight nodes centered around Toolkit::Password Manager, many components in the red and green communities are not included or faded in the background (e.g., Firefox::Theme and WebExtensions::Themes at top-left corner).

The blue community in Fig. 3 is one of the small communities, which includes Toolkit::Password Manager, Firefox::about:logins, Toolkit::Password Manager:Site Compatibility and Toolkit::Form Autofill. This tossing community is the result of complex component design and misleading component name. The first three components are all related to password management (see Mozilla wiki and meeting note). The fourth password management component Firefox::Migration does not appear in this community because it did not have tossing paths from or to these components. Toolkit::Form Autofill is not part of password management. It is in the community because five bugs were erroneously assigned to Toolkit::Form Autofill due to misleading component name, but these bugs were finally tossed to Toolkit::Password Manager.

3) Information Enrichment: Existing bug triaging techniques [1], [2] treat components just as class labels and learn from only correctly assigned bug reports. However, in the bug tossing comments, additional information, such as component descriptions and the contrast of correctly and erroneously assigned bugs, often provides the basis and justification for bug tossing decisions. Therefore, we enrich each component in the bug tossing graph with two types of information: component description, and two sets of bugs (closed bugs versus tossed-

TABLE III: Three Categories of Features

Feature	Dimension
Bug Feature	
a. number of bug summary tokens	1
Component Features	
a. bug tossing community index	1
b. number of closed bugs	1
c. number of tossed-out bugs	1
d. in-degree (unweighted & weighted)	2
e. out-degree (unweighted & weighted)	2
f. degree (unweighted & weighted)	2
Bug-Component Relation Features	
a. sim(bug summary, component name)	2
b. sim(bug summary, component description)	2
c. Top30 sim(bug summary, closed bug summary)	60
d. Top30 sim(bug summary, tossed-out bug summary)	60
e. Percentage(nonzero(sim(bug summary, closed bug summary)))	2
f. Percentage(nonzero(sim(bug summary, tossed-out bug summary)))	2

out bugs). This allows us to model and differentiate confusing bug and component information more effectively.

Component description: Unlike class label which is just a component index, component names reveal the identity of components (e.g., key functionality or concept) in a concise form. As component names are sometimes misleading, we also attach component descriptions which provide more details about the components. Sometimes a component name may contain acronyms, and the full names can generally be found in the component description. In such cases, we keep the original acronym and also expand the acronym into the full name, for example Gecko Media Plugin for GMP. For acronym expansion, we first split the component name into tokens and identify acronyms (i.e., tokens with all capital letters like GMP). Then, we match the letters in an acronym with the word initials in the component description by the same order (i.e., G→M→P). If the concatenated initials of a sequence of words match the acronym, this sequence of words is regarded as the full name of the acronym.

Closed and tossed-out bugs: In this work, we assume if a bug is assigned to a component and closed there, this bug assignment is correct. Otherwise, if the bug is tossed to another component, the bug assignment is erroneous. As shown in Fig. 3, components generally have both closed bugs and tossed-out bugs. Some components (e.g., Toolkit::Password Manager, Toolkit::Themes, Firefox::Security) have high ratios between tossed-out bugs and closed bugs. By contrasting an input bug with closed bugs and tossed-out bugs, the model would learn not only how to make correct bug assignments but also how to not make erroneous assignments.

B. Feature Design

In addition to bug report features, our bug tossing knowledge graph allows us to extract a rich set of features about components and bug-component relationships.

1) Text Preprocessing: Our approach takes as input the summary of a bug report. We assume the minimal representation of bug reports so that the approach can have good generalizability. In addition, we also need to process component names, component descriptions and the summaries of historical bug reports attached to the components. We process these texts as follows. First, we use camel case to split words, and convert all words to lowercase. For example, masterPassword is transformed into two lowercase words "master" and "password". Bug report summaries are short sentences. But component descriptions may be a paragraph of text. We split a paragraph into sentences by punctuations. We perform tokenization and stemming using the NLTK tool.

We encode text in two types of word representations: TF/IDF (Term Frequency/Inverse Document Frequency) and word embedding. TF/IDF is a high-dimensional one-hot vector which can cover all words in a text corpus. For IDF computation, we consider the summaries of all bugs attached to a component as a document. Word embeddings are low-dimensional dense vectors. It requires to define a fixed vocabulary size. Bug reports often contain domain-specific terms or acronyms, which are generally out of the word embedding vocabulary. Rather than regarding these domain-specific terms and acronyms as out-of-vocabulary tokens, which may affect the embedding quality, we use FastText that computes word embeddings at subword level. We multiply the learned word embeddings with IDF as a weight factor.

2) Input Bug Feature: Exist techniques [2], [6] directly use the words in bug report summary as input features. We do not do so because the text often contains misleading keywords that are hard to distinguish from key problem information by looking at the text alone. In our approach, we take only a simple feature of input bug, i.e., the number of tokens in the bug report summary. This feature is indicative of the amount of information in the bug report summary. We make full use of the information in the bug report summary as bug-component relation features detailed in Section III-B4.

3) Component Features: Exist techniques [2], [6] simply treat components as class labels. In contrast, we make good use of rich information that the components have, including three types of nine features as listed in Table III.

Component community: the index of the bug tossing community that a component belongs to. This feature informs the model whether the two components compared during learn to rank come from the same community. The same community indicates that the two components have a track record of being confusing. However, assigning a bug to bystander components in the same community as the resolver component is still better than assigning the bug to the components in other communities. Take Bug 1644112 as an example. If it cannot be assigned to Firefox::about:logins, it would be better to assign it to Toolkit::Password Manager, rather than Core::Graphics:Text, as developers of components in the same community likely know where to toss, but developers may not have expertise to make the right tossing across the communities.

The number of closed and tossed-out bugs: These two features help to judge the probability of a component being the resolver or bystander of a bug and the probability of making mistakes. In Fig 3, we see that Toolkit::Password Manager has 594 closed bugs and 277 tossed-out bugs. Firefox::about:logins has 304 closed bugs and only 10 tossed-out bugs. Comparing these two components, Toolkit::Password Manager is the resolver of much more bugs, but assigning a bug to Toolkit::Password Manager has higher error probability

than assigning it to Firefox::about:logins.

unweighted and weighted (in/out)-degree: These six features represent how many toss-in and toss-out relationships a component has with its direct neighbors. In-degree corresponds to toss-in, out-degree corresponds toss-out, and degree regards the edge as indirected. Unweighted metric counts the number of edges. Weighted metric measures how strong the relationship is, i.e., the sum of the frequencies on the edges. In Fig 3, the in-degree of Toolkit::Password Manager is 8, and the weighted in-degree is 30. The out-degree of Toolkit::Password Manager is 6 and the weighted out-degree is 267. The degree and weighted degree are 14 and 297 respectively.

4) Bug-Component Relation Features: Exist techniques [2], [6] infer implicit bug-component relations through the classifier. In contrast, we explicitly model bug-component relations in terms of the similarities between the input bug summary and the information (component description, closed bugs and tossed-out bugs) of components, as three types of six features. These text similarity features contrast the input bug summary with component information, rather than directly using bug summary text as features. The similarity is computed based on TF/IDF vectors and word embeddings. TF/IDF reflects whether the bug summary and the component information have some keywords in common, while word embeddings measure syntactic and semantic text similarity in a more fuzzy manner. We compute the cosine similarity between the two vectors, and use both TF/IDF-based and word-embedding-based similarities to achieve complementary effects.

Similarity between input bug summary and component name/description: These four features (two for TF/IDF and two for word embedding) allow the model to take into account rich semantic in component names and descriptions and semantic relations between bugs and components.

Top 30 similarities between the input bug and closed bugs: This input-closed feature is a 60-dimensional vector (30 for TF/IDF and 30 for word embedding). Each dimension is a similarity between the input bug summary and the summary of one closed bug of the component. The component may have more than 30 closed bugs. We rank the similarities and take top 30. The input-closed feature informs the model how the input bug is similar to up to 30 closed bugs of the components. Furthermore, they reflect the distribution of the similarities. Consider the Bug 1644807 initially assigned to Firefox::Preferences and then tossed to Toolkit::Password Manager. The top 5 input-closed similarities with Toolkit::Password Manager is $\{$0.803, 0.798, 0.794, 0.789, 0.783$\}$, while the top 5 input-closed similarities with Firefox::Preferences is $\{$0.791, 0.761, 0.739, 0.693, 0.678$\}$. We see that the Bug 1644807 is consistently similar to the closed bugs of Toolkit::Password Manager. In contrast, this bug is only very similar to two closed bugs of Firefox::Preferences, but not the rest of closed bugs. This distribution comparison during learning to rank would help the model make the right choice.

Top 30 similarities between the input bug and tossed-out bugs: This input-tossedout feature contains top 30 similarities (TF/IDF and word embedding respectively) between the input bug and the top-30 most similar tossed-out bugs of the component. The input-tossedout feature, as opposed to the input-closed feature, allows the model to see how the input bug looks like past mistakes. For example, the top 5 input-closed similarities for the Bug 1618597 and the component Firefox::about:logins is $\{$0.826, 0.819, 0.812, 0.808, 0.808$\}$, which make us feel Firefox::about:logins could be the resolver. However, when looking at the top 5 input-tossedout similarities $\{$0.773, 0.737, 0.633, 0.584, 0.584$\}$ for the Bug 1618597 and Firefox::about:logins, we may realize that this assignment could be risky as the bug is very similar to some tossed-out bugs of Firefox::about:logins.

Percentage of nonzero input-closed or input-tossedout similarities: The nonzero-input-closed (or nonzero-input-tossedout) feature computes the percentage of all closed (or tossed-out) bugs of a component that have nonzero similarities with the input bug. For the component with less than 30 closed (or tossed-out) bugs, many dimensions of input-closed (or input-tossedout) will be zero, which makes this feature less effective or even misleading. These two features would inform the model these zero-value dimensions may be useless.

C. Learning to Rank Framework

Considering complex and confusing relations between bugs and components, our approach does not learn from from each bug-component assignment independently. Instead, our approach adopts a learning-to-rank framework that learns from more relevant assignments against less relevant assignments pairwisely. In particular, we adopt LambdaMart [9] to rank components for an input bug.

Learning to rank is a supervised machine learning algorithm. Therefore, preparing meaningful training data is important. Our learning goal is to assign an input bug to the resolver component or at least to some bystander components having tossing relationships with the resolver component. Based on this goal, we construct the training data from bug triaging and tossing history as follows. Let b be a bug and C_f be its resolver. Without losing generality, let $C_1 \rightarrow C_2 \rightarrow C_f$ be a real tossing path. A correct initial assignment without tossing can be regarded as a tossing path with length 0. We build a data point for each component on the tossing path. The bug-component relevance is computed as $\frac{1}{2^{d(C_x,C_f)}}$ where $d(C_x, C_f)$ is the distance from C_x to c_f along the path. In this example, , $d(C_1, C_f)$=2, $d(C_1, C_f)$=1 and $d(C_f, C_f)$=0, and thus the relevance labels for b-C_x (x=1,2,f) are 0.25, 0.5 and 1. For all other components C_o not on the tossing path, we create a data point for each of them with relevance label 0, i.e., b-C_0=0. We construct data points using real tossing paths rather than goal-oriented tossing paths because real tossing paths produce more data points and more fine-grained bug-component relevance labels. For example, for another bug b' with real tossing path $C_1 \rightarrow C_f$, the relevance label for b'-C_1 is 0.5. By goal-oriented tossing path $C_1 \rightarrow C_f$, we will have only one data point with relevance label 0.5 for the two bugs.

For each data point b-C, we associate it with the three categories of features described in Section III-B. At training time, LambdaMart optimizes the gradient of objective function through gradient boosting [9] which can be modeled by the sorted positions of the components for an input bug b against the ground-truth relevance labels. At inference time, give an unseen bug, the trained model is used to produce a relevance label for each component to be predicted based on the features between the bug and the component, and then rank the components by the predicted relevance labels.

IV. Evaluation

This section reports the evaluation of our LR-BKG to answer the following research questions:

- RQ1: Can LR-BKG improve resolver component recommendation over traditional bug triaging methods?
- RQ2: Can LR-BKG improve tossing-relevant component (resolver and bystander) recommendation over traditional bug triaging methods?
- RQ3: What features are the most important for the performance of our approach?

A. Experimental Setup

1) Dataset: We crawl 124,477 bugs of six Mozilla products and their 186 components. To ensure the validity of correct bug-component assignments, we retain only bugs whose status is resolved, verified or closed. According to Mozilla's bug status guide, bugs with these statuses are closed Bugs. We assume that the component of a closed bug is the bug's resolver. For bugs with other statuses, it is still uncertain which components would be their resolvers. After filtering by closed bugs, we obtain 98,587 closed bugs covering 6 products and 186 components. 29,100 out of these 98,587 closed bugs are tossed bugs. We split the dataset into training data and testing data. In order to simulate real-world context, we cut dataset into 80% and 20% in chronological order. 78,870 bugs (including 24,039 tossed bugs) with creation time before 25th February, 2020 are "historical" training data. The rest 19,717 bugs (including 5,061 tossed bugs) are "future" testing data.

2) Baseline: We compare our approach with two baselines. This first baseline is the BugBug tool (in particular its component classifier for assigning bugs to product::component) [6]. The BugBug tool has been actively used in the Mozilla's product development. It formulates bug triaging as a multi-class classification task, and uses the logistic regression model built in XGBoost. XGBoost is an optimized distributed gradient boosting library, which implements machine learning algorithms under gradient boosting framework [10]. BugBug takes the one-hot representation of bug summary, description and keywords/flags as features. We train BugBug using our historical bug dataset.

The second baseline is BugBug with tossing graph (BugBug-TG). This baseline is inspired by Jeong et al. [7] which uses bug tossing graph to improve the initial recommendation lists by any bug triaging classifiers. Let $\{C_1, C_2, ...C_n\}$ be the initial recommendation list. BugBug-TG starts from the top, and inserts the most possible tossing target C_{xt} of each component C_x based on the historical tossing probability in the bug tossing graph. The resulting list looks like $\{C_1, C_{1t}, C_2, C_{2t}, ...C_n, C_{nt}\}$. Finally, BugBug-TG returns the top N elements in the resulting list.

TABLE IV: Top-k Accuracy

Tool	Category	Top-1	Top-3	Top-5	Top-10
Our Approach	Tossed	0.469	0.701	0.772	0.848
	Non-Tossed	0.593	0.779	0.836	0.892
	Overall	0.562	0.759	0.820	0.881
BugBug	Tossed	0.378	0.608	0.680	0.760
	Non-Tossed	0.468	0.642	0.697	0.764
	Overall	0.445	0.633	0.692	0.763
BugBug with TG	Tossed	0.378	0.610	0.680	0.760
	Non-Tossed	0.468	0.644	0.698	0.765
	Overall	0.445	0.635	0.694	0.764

TABLE V: Component-Level Top-k Accuracy Comparison

	Top-1	Top-3	Top-5	Top-10	
#P::C (our<BugBug)	22	13	7	5	28 (any <)
#P::C (our=BugBug)	28	22	24	22	
#P::C (our>BugBug)	136	151	155	159	120 (all >)

3) Evaluation Metrics: For RQ1, we evaluate if a method can recommend the resolver component for a bug. As there is only one resolver component for a bug, we use Top-k accuracy (also known as Hit@k accuracy) to evaluate the performance of resolver component recommendation. Top-k accuracy is $\sum_{\mathrm{b_i \in B}} \mathrm{isCorrect}(b_i, \mathrm{Top\text{-}k})/|B|$ where B represents the set of all test bugs and the $\mathrm{isCorrect}(b_i, \mathrm{Top\text{-}k})$ returns 1 if Top-k components contain the resolver component of the input bug b_i, and returns 0 otherwise.

For RQ2, we want to recommend not only the resolver component but also bystander components. We consider two types of bystanders: the components in the same tossing community as the resolver component, and the components having direct edges to the resolver component in the goal-oriented bug tossing graph. In addition to Top-k accuracy, we compute NDCG@k in RQ2. $NDCG@k$ is $DCG@k/IDCG@k$, and $DCG@k$ is $\sum_{i=1}^{k} r_i/\log_2(i+1)$ where $r_i = 1$ if the i-th component in the recommendation list is related (resolver or bystander) to the input bug, and $r_i = 0$ otherwise. IDCG is the ideal result of DCG, which means all related components are ranked higher than unrelated ones.

For RQ3, we use feature importance analysis provided in XGBoost [10] to validate the effectiveness of our feature design. Feature importance analysis plots importance based on fitted trees. The importance is calculated by importance types, either "weight", "gain", or "cover"."weight" is the number of times a feature appears in a tree. "gain" is the average gain of splits which use the feature. "cover" is the average coverage of splits which use the feature and the coverage is defined as the number of samples affected by the split. The importance type we use in this study is "weight".

B. Resolver Component Recommendation (RQ1)

Table IV shows the results. We can see that BugBug and BugBug-TG have almost the same accuracies at all Top-ks.

This means that BugBug-TG's simple heuristic use of component tossing relationships is not effective. In contrast, our LR-BKG achieves significant higher accuracies than BugBug and BugBug-TG at all Top-ks. At Top-1, LR-BKG achieves overall 0.562 accuracy, 0.469 for tossed bugs and 0.593 for non-tossed bugs, while BugBug achieves only 0.445 overall, 0.378 for tossed bugs and 0.468 for non-tossed bugs. At Top-10, LR-BKG achieves overall 0.88 accuracy, while BugBug is only 0.76. Although our initial goal focuses on tossed bugs, our LR-BKG actually achieves the same-level improvement for both tossed and non-tossed bugs.

Consider a tossed bug Bug 1644807 which was initially assigned to Firefox::Preferences and tossed to Toolkit::Password Manager. BugBug recommends Firefox::Preferences at the top 1 position. The summary of Bug 1644807 states "Replace all user-facing instances that refer to master password". The keywords such as user, replace, master password are highly related to Firefox::Preferences and its historical bugs. These keywords taken directly as features in BugBug confuse the classifier. In contrast, our LR-BKG correctly recommends the resolver component Toolkit::Password Manager at top 1.

Consider a non-tossed bug Bug 1618597 whose resolver component is Toolkit::Password Manager. However, BugBug recommends Firefox::about:logins at top 1 which could mislead the developers. Our LR-BKG recommends Toolkit::Password Manager at top 1. As seen in Fig. 3, Toolkit::Password Manager and Firefox::about:logins are very confusing components which have many erroneous assignments and tossing in between. The summary of Bug 1618597 is "Saved logins and Master Password are cleared when upgrading from Firefox56 to Firefox74.0beta via Firefox73.0.1" which includes many confusing keywords such as saved, logins and master password. These keywords are rather common in the closed bugs of both Toolkit::Password Manager and Firefox::about:logins. However, our approach can effectively distinguish key problem information (e.g., cleared, upgrade) from these common confusing information, and consequently make the right recommendation.

We compute the Top-k accuracy for each component by LR-BKG and BugBug. We count for how many components LR-BKG achieves higher, equal or lower accuracies than BugBug. We consider the accuracy difference below 0.03 as equal. Table V presents the results. The last column means if LR-BKG is lower than BugBug at any Top-k, we count it as LR-BKG<BugBug. Only if LR-BKG is higher than BugBug at all Top-k, we count it as LR-BKG>BugBug.

At different Top-ks, LR-BKG achieves higher accuracies than BugBug for 73%-85% of 186 components, and achieves the same accuracies as BugBug for 12%-15% of components, and achieves lower accuracies for only 3%-12% of components. With the strictest comparison, LR-BKG is better than BugBug for 120 (65%) components (all>), and is worse than BugBug for only 28 (15%) components (any<). Consider Core::Networking:Cookies that has 30 tossed bugs. BugBug makes 13 correct recommendation at top 1 and 18 in top 10. The erroneous recommendations includes not only confusing components (e.g, Core::Security:PSM, Core::Networking:HTTP) but also some non-sense components (e.g., DevTools::Console). LR-BKG recommends correct resolver components for 18 tossed bugs at top 1 and for 24 tossed bugs in top 10.

We find that LR-BKG sometimes makes mistakes for resolver components in bug tossing communities. For example, Firefox::Sync belongs to the community that includes 48 components (the red community in Fig. 3). Firefox::Sync has 140 bugs, among which LR-BKG assigns 52 bugs to other components in the community, such as Firefox::Tabbed Browser, Firefox::Preferences. This results in its low Top-1 accuracy 0.443 which is lower than BugBug (0.543). However, LR-BKG does not rank resolver components totally off the track. For the Top-3, Top-5 and Top-10 accuracies, LR-BKG is 0.764, 0.836 and 0.900 respectively, while BugBug is only 0.707, 0.743 and 0.771. That is, LR-BKG still identifies the resolver component, but it has some difficulties in distinguishing the resolver components from some bystander components in the large community at the top-1 position.

> Our LR-BKG significantly improves the accuracy of resolver component recommendation. The improvement comes from both tossed and non-tossed bugs equally, and is contributed by the improvement on the majority of components.

C. Tossing-Relevant Component Recommendation (RQ2)

Table VIa to Table VId show the results. Comparing the results in Table IV, Table VIa and Table VIc, we see that all approaches can recommend some bystander components. For example, the overall Top-1 accuracy for resolver component by LR-BKG is 0.562 (Table IV). If we consider the bystander components from the same bug tossing community as correct recommendation, the overall Top-1 accuracy by LR-BKG becomes 0.799 (Table VIa). Furthermore, many bystander components can be ranked in top 10 which gives > 0.9 Top-10 accuracies for all three methods.

As Top-k increases, the accuracy gap between our approach and BugBug narrows. However, the accuracy at Top-1 by our approach is much higher than that of BugBug for both bug tossing community and direct edge settings. For example, Bug 1644112 happens to be in our "future" testing data. BugBug recommends Core::Graphics:Text at top 1 for this bug. Although our approach does not recommend Firefox::about:logins at top 1 either, it recommends Toolkit::Password Manager from the same bug tossing community at top 1. The developers of Toolkit::Password Manager more likely know where to toss Bug 1644112 than the developers of Core::Graphics:Text.

Comparing NDCG@k of different approaches, we see that our approach can better rank tossing-relevant components at the top of the list. Consider the bug tossing community setting. Our approach and BugBug have very close Top-10 accuracy (less than 0.02 difference), but our approach achieves NDCG@10=0.590 while BugBug's NDCG@10 is only 0.432. Similar to resolver component recommendation,

(a) Top-k Accuracy (Bug Tossing Community)

Tool	Category	Top-1	Top-3	Top-5	Top-10
Our Approach	Tossed	0.798	0.917	0.954	0.981
	Non-Tossed	0.799	0.919	0.951	0.976
	Overall	0.799	0.919	0.952	0.977
BugBug	Tossed	0.600	0.865	0.920	0.966
	Non-Tossed	0.615	0.782	0.863	0.931
	Overall	0.612	0.803	0.878	0.940
BugBug with TG	Tossed	0.600	0.860	0.917	0.965
	Non-Tossed	0.615	0.780	0.861	0.930
	Overall	0.612	0.801	0.876	0.939

(b) NDCG@k (Bug Tossing Community)

Tool	Category	ndcg@1	ndcg@3	ndcg@5	ndcg@10
Our Approach	Tossed	0.798	0.712	0.665	0.590
	Non-Tossed	0.799	0.684	0.630	0.563
	Overall	0.799	0.691	0.639	0.570
BugBug	Tossed	0.600	0.534	0.491	0.432
	Non-Tossed	0.615	0.485	0.438	0.385
	Overall	0.612	0.497	0.451	0.397
BugBug with TG	Tossed	0.600	0.537	0.491	0.431
	Non-Tossed	0.615	0.494	0.445	0.390
	Overall	0.612	0.505	0.457	0.401

(c) Top-k Accuracy (Direct Edge)

Tool	Category	Top-1	Top-3	Top-5	Top-10
Our Approach	Tossed	0.699	0.887	0.935	0.974
	Non-Tossed	0.761	0.902	0.939	0.969
	Overall	0.745	0.898	0.938	0.970
BugBug	Tossed	0.549	0.818	0.891	0.952
	Non-Tossed	0.592	0.765	0.845	0.920
	Overall	0.581	0.778	0.856	0.928
BugBug with TG	Tossed	0.549	0.816	0.886	0.950
	Non-Tossed	0.592	0.764	0.843	0.919
	Overall	0.581	0.777	0.854	0.927

(d) NDCG@k (Direct Edge)

Tool	Category	ndcg@1	ndcg@3	ndcg@5	ndcg@10
Our Approach	Tossed	0.699	0.599	0.547	0.497
	Non-Tossed	0.761	0.630	0.574	0.521
	Overall	0.745	0.622	0.567	0.515
BugBug	Tossed	0.549	0.466	0.430	0.400
	Non-Tossed	0.592	0.455	0.409	0.385
	Overall	0.581	0.458	0.414	0.389
BugBug with TG	Tossed	0.549	0.470	0.431	0.400
	Non-Tossed	0.592	0.464	0.416	0.390
	Overall	0.581	0.465	0.420	0.393

TABLE VI: Results of RQ2

BugBug and BugBug-TG have almost the same accuracies and NDCG metrics, which means simple heuristic use of component tossing relationships is not effective for improving bystander component recommendation.

> Our LR-BKG can significantly improve the ranking of bystander components that have historical tossing relationships with the resolver component.

D. Feature Importance (RQ3)

We estimate the importance of 138 features (see Table VII) by feature importance analysis [10], which assigns important scores to input features based on how useful they are at predicting a target variable. Table VII presents the top 30 important features, which include one bug feature, six component features and 23 bug-component relation features.

TABLE VII: Top-30 Important Features

Feature	Ranking
Bug Feature	
a. number of bug summary tokens	7
Component Features	
a. number of closed bugs	8
b. weighted in-degree	9
c. bug tossing community index	18
d. unweighted in-degree	23
e. unweighted degree	27
f. unweighted out-degree	30
Bug-Component Relation Features	
a. TFIDF sim(bug summary, component name)	21
b. TFIDF sim(bug summary, component description)	14
c. FASTTEXT sim(bug summary, component name)	12
d. FASTTEXT sim(bug summary, component description)	16
e. TFIDF Top 1-7, 10, 8 sim(bug summary, closed bug summary)	1, 5, 10, 11, 13, 17, 19, 22, 26
f. TFIDF Percentage(nonzero(sim(bug summary, closed bug summary)))	2
g. TFIDF Top 1-4 sim(bug summary, tossed-out bug summary)))	4, 15, 20, 28
h. TFIDF Percentage(nonzero(sim(bug summary, tossed-out bug summary)))	24
i. FASTTEXT Top 1 sim(bug summary, closed bug summary)	6
j. FASTTEXT Top 1, 2, 4 sim(bug summary, tossed-out bug summary)))	3, 25, 29

The number of bug summary tokens is the only bug feature we use. This simple feature is ranked at the 7th importance. Six out of nine component features are important. The number of closed bugs is ranked at the 8th position. Bug tossing community index is ranked at 18th. Weighted in-degree and three unweighted degree features are important, ranked at the 9th, 23rd, 27th and 30th positions. For bug-component relation features, both TF/IDF-based and word-embedding-based similarities between input bug summary and component name/description are important. TF/IDF similarities are more important than word-embedding similarities for measuring the relevance of the input bug to the closed bugs of the components. TF/IDF and word-embedding similarities are equally important for measuring the relevance of the input bug to the tossed-out bugs of the components. We see that the model mainly focused on the top-ranked similarities. It considers more top-ranked input-closed similarities (up to top 10) than input-tossedout similarities (up to top 4). Non-zero similarity percentage features are important to inform the model the usefulness of input-closed and input-tossed-out feature vectors.

> All categories of features play important role, which demonstrates the effectiveness of our feature design. Input-closed and input-tossedout features could be simplified.

V. Related Work

Machine learning techniques have been widely adopted to support bug report management [3], including bug field prediction, bug field reassignment, bug localization, duplicate bug detection, bug-commit linking, etc. The closed work to ours is bug field prediction. Anvik et al. [5] propose to use the

classifier (e.g., Support Vector Machine) to assist bug fixer assignment. Other studies focus on predicting bug severity [11]–[14] and bug priority [15], [16]. Several studies [1], [2] investigate component assignments. Somasundaram et al. [1] mine the topic model from the component's historical bugs for measuring bug component relevance. Sureka [2] uses the Navie Bayes classifier with TF/IDF and dynamic language model to predict the component of a bug report. The classification-based method has been adopted in practice, e.g., the Mozilla's BugBug tool. Recently, deep-learning based classification has also been used for supporting incident triage for online service systems [17]. Different from these classification-based methods, our approach is based on learning-to-rank, which learns not only from correct bug-component assignments but also erroneous and irrelevant assignments.

Jeong et al. [7] models developer tossing relationships in a probability graph which helps to uncover team structures and recommend bug fixers. Our work is different in that we construct a bug tossing graph among components and learn from historical component tossing relationships to reduce bug-component assignment confusion. Our baseline BugBug-TG is inspired by the way that Jeong et al. [7] use the developer tossing graph for improving developer recommendation, but BugBug-TG is based on our component tossing graph. Our results show that the simple heuristic use of component tossing relationships is not effective for improving bug component assignment. Bhattacharya et al. [18], [19] extend Jeong et al. [7] by incremental learning for effective use of each training data point. Hu et al. [20] analyzes historical bug-component and bug-fixer relations to assist fixer recommendation. However, none of these works effectively use erroneous and irrelevant assignments during learning.

Xia et al. [4] report an empirical study on bug field reassignment and they find that fixer and component reassignments occur most frequently. Their follow-up work [21] trains multi-label classifiers to predict the possibility of field reassignment. Similar approaches have been adopted in [22]–[24]. However, these works do not predict which specific components should be reassigned to. Tian et al. [25] and Han et al. [26] address the problem of developer reassignment. They also use learning-to-rank, but their feature design is developer-centric. In contrast, our feature design is component-centric, with several explicit features for component tossing community and tossed-out bugs. We also explicitly model and contrast the similarity distributions between the input bug and the closed/tossed-out bugs of the components. These features are inspired by our empirical observation on additional information and mechanisms developers often resort to for justifying bug tossing decisions.

VI. Conclusions and Future Work

This paper presents an empirical study on bug tossing phenomena and root cause, and a learning to rank framework to address the limitations of classification-based bug component assignment revealed in our empirical study. Different from classification-based methods that learn from each bug-component assignment independently, learning-to-rank learns by contrasting correct, erroneous and irrelevant assignments. Inspired by additional information developers often resort to for correcting erroneous bug assignments, we construct a bug tossing knowledge graph from which rich bug, component and bug-component relation features can be derived and used to inform the learning-to-rank model. Our large-scale evaluation confirms the effectiveness of our learning-to-rank approach and feature design. In the future, we will enrich our bug tossing knowledge graph (e.g., with bug topics), improve our feature design (e.g., on component community and bug similarity distribution), and experiment deep-learning based feature extraction in the learning-to-rank framework.

VII. Acknowledgement

This research was partially funded by Data61-ANU Collaborative Research Project CO19314. We sincerely thank the anonymous reviewers for the insightful and constructive feedback.

References

[1] K. Somasundaram and G. C. Murphy, "Automatic categorization of bug reports using latent dirichlet allocation," in *Proceeding of the 5th Annual India Software Engineering Conference, ISEC 2012, Kanpur, India, February 22-25, 2012*, S. K. Aggarwal, T. V. Prabhakar, V. Varma, and S. Padmanabhuni, Eds. ACM, 2012, pp. 125–130. [Online]. Available: https://doi.org/10.1145/2134254.2134276

[2] A. Sureka, "Learning to classify bug reports into components," in *Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings*, ser. Lecture Notes in Computer Science, C. A. Furia and S. Nanz, Eds., vol. 7304. Springer, 2012, pp. 288–303. [Online]. Available: https://doi.org/10.1007/978-3-642-30561-0_20

[3] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu, "How practitioners perceive automated bug report management techniques," *IEEE Trans. Software Eng.*, vol. 46, no. 8, pp. 836–862, 2020. [Online]. Available: https://doi.org/10.1109/TSE.2018.2870414

[4] X. Xia, D. Lo, M. Wen, E. Shihab, and B. Zhou, "An empirical study of bug report field reassignment," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, S. Demeyer, D. W. Binkley, and F. Ricca, Eds. IEEE Computer Society, 2014, pp. 174–183. [Online]. Available: https://doi.org/10.1109/CSMR-WCRE.2014.6747167

[5] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 361–370. [Online]. Available: https://doi.org/10.1145/1134285.1134336

[6] S. L. Marco Castelluccio, "bugbug," GitHub, https://github.com/mozilla/bugbug.

[7] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, H. van Vliet and V. Issarny, Eds. ACM, 2009, pp. 111–120. [Online]. Available: https://doi.org/10.1145/1595696.1595715

[8] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.

[9] C. J. Burges, "From ranknet to lambdarank to lambdamart: An overview," *Learning*, vol. 11, no. 23-581, p. 81, 2010.

[10] "xgboost," GitHub, https://github.com/dmlc/xgboost.

[11] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China*. IEEE Computer Society, 2008, pp. 346–355. [Online]. Available: https://doi.org/10.1109/ICSM.2008.4658083

[12] T. Zhang, J. Chen, G. Yang, B. Lee, and X. Luo, "Towards more accurate severity prediction and fixer recommendation of software bugs," *J. Syst. Softw.*, vol. 117, pp. 166–184, 2016. [Online]. Available: https://doi.org/10.1016/j.jss.2016.02.034

[13] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*, J. Whitehead and T. Zimmermann, Eds. IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: https://doi.org/10.1109/MSR.2010.5463284

[14] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*. IEEE Computer Society, 2012, pp. 215–224. [Online]. Available: https://doi.org/10.1109/WCRE.2012.31

[15] ——, "DRONE: predicting priority of reported bugs by multi-factor analysis," in *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. IEEE Computer Society, 2013, pp. 200–209. [Online]. Available: https://doi.org/10.1109/ICSM.2013.31

[16] F. Khomh, B. Chan, Y. Zou, and A. E. Hassan, "An entropy evaluation approach for triaging field crashes: A case study of mozilla firefox," in *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, M. Pinzger, D. Poshyvanyk, and J. Buckley, Eds. IEEE Computer Society, 2011, pp. 261–270. [Online]. Available: https://doi.org/10.1109/WCRE.2011.39

[17] J. Chen, X. He, Q. Lin, H. Zhang, D. Hao, F. Gao, Z. Xu, Y. Dang, and D. Zhang, "Continuous incident triage for large-scale online service systems," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 364–375. [Online]. Available: https://doi.org/10.1109/ASE.2019.00042

[18] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," in *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, R. Marinescu, M. Lanza, and A. Marcus, Eds. IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: https://doi.org/10.1109/ICSM.2010.5609736

[19] P. Bhattacharya, I. Neamtiu, and C. R. Shelton, "Automated, highly-accurate, bug assignment using machine learning and tossing graphs," *J. Syst. Softw.*, vol. 85, no. 10, pp. 2275–2292, 2012. [Online]. Available: https://doi.org/10.1016/j.jss.2012.04.053

[20] H. Hu, H. Zhang, J. Xuan, and W. Sun, "Effective bug triage based on historical bug-fix information," in *Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering*, ser. ISSRE '14. USA: IEEE Computer Society, 2014, p. 122–132. [Online]. Available: https://doi.org/10.1109/ISSRE.2014.17

[21] X. Xia, D. Lo, E. Shihab, and X. Wang, "Automated bug report field reassignment and refinement prediction," *IEEE Trans. Reliab.*, vol. 65, no. 3, pp. 1094–1113, 2016. [Online]. Available: https://doi.org/10.1109/TR.2015.2484074

[22] A. Lamkanfi and S. Demeyer, "Predicting reassignments of bug reports - an exploratory investigation," in *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, A. Cleve, F. Ricca, and M. Cerioli, Eds. IEEE Computer Society, 2013, pp. 327–330. [Online]. Available: https://doi.org/10.1109/CSMR.2013.42

[23] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K. Matsumoto, "Studying re-opened bugs in open source software," *Empir. Softw. Eng.*, vol. 18, no. 5, pp. 1005–1042, 2013. [Online]. Available: https://doi.org/10.1007/s10664-012-9228-6

[24] ——, "Predicting re-opened bugs: A case study on the eclipse project," in *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, G. Antoniol, M. Pinzger, and E. J. Chikofsky, Eds. IEEE Computer Society, 2010, pp. 249–258. [Online]. Available: https://doi.org/10.1109/WCRE.2010.36

[25] Y. Tian, D. Wijedasa, D. Lo, and C. L. Goues, "Learning to rank for bug report assignee recommendation," in *24th IEEE International Conference on Program Comprehension, ICPC 2016, Austin, TX, USA, May 16-17, 2016*. IEEE Computer Society, 2016, pp. 1–10. [Online]. Available: https://doi.org/10.1109/ICPC.2016.7503715

[26] J. Han, J. Li, and A. Sun, "UFTR: A unified framework for ticket routing," *CoRR*, vol. abs/2003.00703, 2020. [Online]. Available: https://arxiv.org/abs/2003.00703