Learning to Rank for Bug Report Assignee Recommendation

Yuan Tian*, Dinusha Wijedasa*, David Lo*, and Claire Le Goues†
*School of Information Systems, Singapore Management University, Singapore
†School of Computer Science, Carnegie Mellon University, USA
{yuan.tian.2012,dwijedasa,davidlo}@smu.edu.sg, clegoues@cs.cmu.edu

***Abstract*—Projects receive a large number of bug reports, and resolving these reports take considerable time and human resources. To aid developers in the resolution of bug reports, various automated techniques have been proposed to identify and recommend developers to address newly reported bugs. Two families of bug assignee recommendation techniques include those that recommend developers who have fixed similar bugs before (a.k.a. activity-based techniques) and those recommend suitable developers based on the location of the bug (a.k.a. location-based techniques). Previously, each of these techniques has been investigated separately.**

In this work, we propose a unified model that *combines* information from both developers' previous activities and suspicious program locations associated with a bug report in the form of *similarity features*. We have evaluated our proposed approach on more than 11,000 bug reports from Eclipse JDT, Eclipse SWT and ArgoUML projects. Our experiments show that our unified model can outperform a location-based baseline by Anvik et al. and an activity-based baseline by Shokripour et al. In terms of correct recommendations at top-1 position, our unified model outperforms the activity-based baseline 50.0%-100.0%, and the location-based baseline by 11.1%-27.0%.

***Index Terms*—Bug Reports; Bug Assignee Recommendation**

I. Introduction

Software projects use bug tracking systems, such as Bugzilla and JIRA, to access and manage change requests or bugs that have been reported. Users, developers, and testers can leverage such systems to easily submit a bug or a feature request as a formatted bug report. Project maintainers can then use the system to validate reports and manage the resolution process.

Although bug tracking systems help developers, bug resolution still takes significant time and human resources. This is especially the case for large projects with large number of bug reports. For instance, during September 2015, the Eclipse project received approximately 2,500 bug reports, averaging 80 new reports each day.[1] These large quantities of daily reported bugs must be *triaged* by project maintainers (or triagers) to assign each issue to an appropriate developer/team for handling. This process is important, because assigning a bug report to the incorrect developer or team can increase the overall time required to fix the bug, increasing project maintenance cost [1]. Moreover, this process is time consuming and non-trivial since good comprehension of bug report, source code, and team members is needed.

To improve bug triage efficiency and effectiveness, researchers have proposed numerous approaches to automatically assign bug reports to developers, by extracting useful information from historical bug reports and source code [1], [2], [3], [4], [5], [6]. Shokripour et al. categorized this prior work into two groups, based on their underlying mechanism: activity-based approaches, and location-based approaches [1]. *Activity-based* approaches [2], [3], [4] identify potentially appropriate developers based on their activities (e.g., previously fixed bugs) within the project, across various project artifacts. By contrast, *location-based* approaches [1], [5], [6] recommend a bug report assignment by localizing the bug to a set of potential source code locations and then identifying which developers touched the implicated code. Each approach has its pros and cons. For instance, location-based approaches are highly reliant on the performance of bug localization, which might not be high (c.f. [7], [8], [9], [10]); activity-based approaches might be inappropriately biased by the previous activities of a given developer. We discuss these limitations in more detail in Section II. Importantly, none of the previous work has combined the two types of information, which motivates our study.

In this work, we propose a unified model (based on learning to rank machine learning technique) that *combines* information from both developers' previous activities and suspicious program locations associated with a bug report in the form of *similarity features*. Learning to rank is a machine learning technique widely applied in applications like document retrieval. We choose learning to rank because we can map the assignee recommendation task to a document retrieval task by treating the bug report as the query, and developers' profiles (previously fixed bugs and committed source code) as documents to be returned. To incorporate location information, the query can be enriched with the potential locations where the bug may reside. This reduces the task to ranking documents (developers) based on the similarity between a query (bug report) and each document (developer profile).

To capture the similarity between a bug report and developer profile, we propose 16 features, considering both the potential location of the bug (location-based features) and previously fixed bugs by each developer (activity-based features). To evaluate our approach, we collect more than 11,000 bug

[1]We calculated this number by looking at the number of bug reports submitted from September 1st to 31st, 2015, based on the information available at https://bugs.eclipse.org/bugs/

reports together with committed source code from three open source projects: Eclipse JDT, Eclipse SWT and ArgoUML. Our experiments show that combining these two types of features improves the performance of learning to rank model as compared to one that uses only one type of feature. The experiments also show that our unified model achieves better results as compared to a location-based baseline by Anvik et al. [3] and an activity-based baseline by Shokripour et al. [1].

Our key contributions are thus:

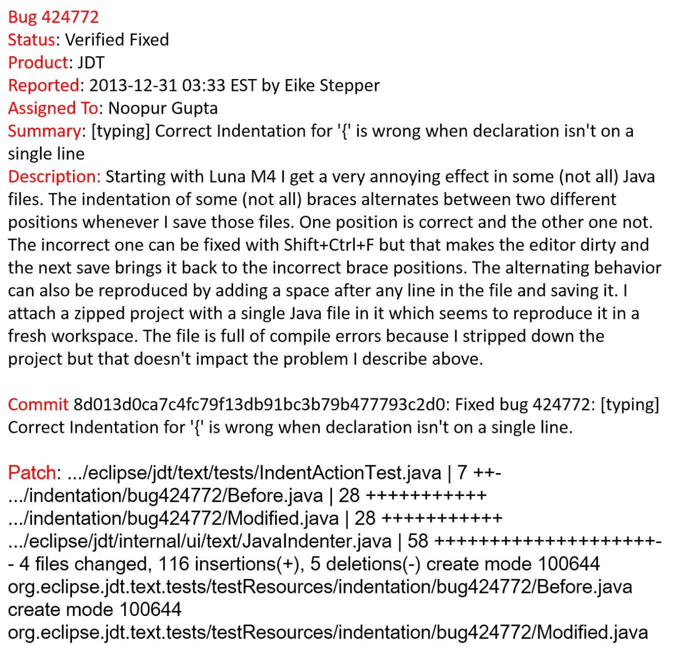
1) A novel unified model based on learning to rank machine learning algorithm. This unified model can leverage information from both developers' activities and the result of bug report localization task. This integrates activity-based and location-based bug assignee recommendation approaches.
2) 16 features to capture the degree a developer matches a bug report.
3) Experimental results on more than 11,000 bugs from three open source projects. The results show that combining location-based features and activity-based features through learning to rank technique can improve the performance of using only one type of feature.

The rest of this paper is organized as follows. First, we provide background on prior bug assignee recommendation approaches in Section II. In Section III, we present our approach, including the ranking model, the definition of features, and how we collect these features. We describe our experimental setup and process in Section IV, and present and analyze the results in Section V. Related work is presented in Section VI. Finally, we conclude and mention future work in Section VII.

II. Background

In this section, we first introduce basic background on the bug assignee recommendation task. Next, we summarize the two types of automatic bug assignee recommendation approaches that have been considered in prior work. To illustrate this discussion, consider the sample bug report taken from the Bugzilla report database for Eclipse (ID 424772), shown in Figure 1. Bugzilla provides several fields to help describe and manage a bug. In Figure 1, we list six fields of particular interest to the bug assignee recommendation task: bug **Status**, the **Product** in which the bug appears, the time at which the bug was **Reported**, the developer to whom the bug was assigned (**Assigned To**), and both a short **Summary** and a long **Description** to provide details and describe steps for reproduction. This report in particular describes a typing related bug under text component of JDT. We also know that this bug was fixed by a developer named Noopur Gupta through a commit, ID of which starts with "8d013d".

When a bug report like this arrives in the system, it usually does not have an assigned developer (like Noopur Gupta). It is the task of the triager or project maintainers to analyze such a report, establish its validity and uniqueness, and then identify the appropriate person or team to address it. This process is manual and time-consuming, given the hundreds of reports large projects receive daily. Our goal in this work is to automate the process of identifying the appropriate developer to whom such a new, valid report should be assigned. The approaches in previous work extract information from such a report and associated project activity recorded in a bug tracking or source control system to construct predictive models. They can be categorized according to the type of information they use: activity-based approaches (see Section II-A) rely on developer activities across various artifacts, linked primarily to textual features in the bug report, while location-based approaches (see Section II-B) use the bug report to locate potentially defective source code files, to identify the developers strongly associated with that code (e.g., developers who created the file, developers who modified the file, developers who modified similar files, etc.).


Bug 424772
Status: Verified Fixed
Product: JDT
Reported: 2013-12-31 03:33 EST by Eike Stepper
Assigned To: Noopur Gupta
Summary: [typing] Correct Indentation for '{' is wrong when declaration isn't on a single line
Description: Starting with Luna M4 I get a very annoying effect in some (not all) Java files. The indentation of some (not all) braces alternates between two different positions whenever I save those files. One position is correct and the other one not. The incorrect one can be fixed with Shift+Ctrl+F but that makes the editor dirty and the next save brings it back to the incorrect brace positions. The alternating behavior can also be reproduced by adding a space after any line in the file and saving it. I attach a zipped project with a single Java file in it which seems to reproduce it in a fresh workspace. The file is full of compile errors because I stripped down the project but that doesn't impact the problem I describe above.

Commit 8d013d0ca7c4fc79f13db91bc3b79b477793c2d0: Fixed bug 424772: [typing] Correct Indentation for '{' is wrong when declaration isn't on a single line.

```
Patch: .../eclipse/jdt/text/tests/IndentActionTest.java | 7 ++-
.../indentation/bug424772/Before.java | 28 +++++++++++
.../indentation/bug424772/Modified.java | 28 +++++++++++
.../eclipse/jdt/internal/ui/text/JavaIndenter.java | 58 ++++++++++++++++++++++-
- 4 files changed, 116 insertions(+), 5 deletions(-) create mode 100644
org.eclipse.jdt.text.tests/testResources/indentation/bug424772/Before.java
create mode 100644
org.eclipse.jdt.text.tests/testResources/indentation/bug424772/Modified.java
```

Fig. 1: Bug report #424772 from Eclipse JDT.

A. Activity-based Bug Assignee Recommendation

Activity-based approaches recommend a developer for a particular bug report based on how well the developer's expertise is predicted to match with the given bug report. Developer expertise is inferred from developer activities during previous bug triage processes [3], [4] and then linked to the words that appear in a new bug report. Consider the bug report shown in Figure 1. The text in the summary and description fields indicates that the problem lies in the JDT Text component, using words like "Indentation", "typing", "braces", "position". Searching the bug database for the JDT project for the keyword "Indentation" reveals several previously fixed bugs related to this concept, and find that many of them were fixed by/assigned to Noopur, the developer that addressed this new bug. For example, Noopur was also assigned to Bug #404821,[2] which reported that the "Code Indentation" feature of the JDT did not work. This illustrates that a report's description and

[2]https://bugs.eclipse.org/bugs/show_bug.cgi?id=404821

summary can provide useful textual information to suggest developers with expertise in a given problem or concept, based on previously fixed bugs.

Previous researchers leveraged this insight in several ways. For instance, Anvik et al. [3] treat the developer as a class label, and bug assignee recommendation task as a multi-class classification problem. They extract features from a set of bug bug reports, i.e., words that appear in the description and summary field of bug report, and represent each bug report as a feature vector. The value of a feature is the number of times a particular word appears in the report. These feature vectors, together with a set of known assignees drawn from previously-addressed bugs, are used as input to learn a predictive model using a classification algorithm (e.g., Support Vector Machine, Naive Bayes Classifiers, Decision Tree). When a new bug report arrives, a similar feature extraction process is applied, and the trained predictive model can be applied on the new feature vector to predict who should fix it.

Although activity-based bug assignee recommendation approaches have been shown to achieve acceptable results, they ignore a valuable source of information, namely the link between bug reports and source code files. This information is leveraged by location based bug assignee recommendation.

B. Location Based Bug Assignee Recommendation

The underlying idea behind location-based bug assignee recommendation approach is to indentify potential developer expertise on the bug report through the source code itself [1], [5], [6]. The basic assumption is that developers who have recently fixed a bug in a given source code file are more likely to have the required expertise to fix a new bug in the same location than other developers. Under this assumption, a developer, even one who has been less active in previous bug fixing activities, has substantial expertise in recently-touched or modified code in the repository.

Generally, these approaches consistent of two phases: (1) bug report localization, followed by (2) bug assignee recommendation. For each of the phases, researchers have proposed various approaches. Hossen et al. apply an information retrieval technique, i.e, latent semantic indexing (LSI) [11] to compute the similarity between a given source file and a bug report [6]. They consider words appearing in identifiers and comments extracted from a source code file as an input document and words appearing in the summary and description field of a bug report as a query. Different from Hossen et al., Shokripour et al. compute a relevance score between a bug report and a source code file by summing the weights of each noun that is common between the bug report and file [1]. The weight of a noun is determined based on the number of times the noun appears in a bug report, a commit message, a source code comment and an identifier. For instance, to fix the sample bug shown in Figure 1, Noopur committed several source files including one named "/eclipse/jdt/internal/ui/text/JavaIndenter.java". When Shokripour et al. compute the similarity between the sample bug and file "JavaIndenter.java", the word "indentation" has a weight of 3, because it appeared in three information sources, the commit message, the identifiers in the file, and the bug report.

Although location-based approaches consider similarities between bug reports and source code files, which activity based approaches ignore, they have drawbacks:

- **High dependence on an underlying bug localization technique.** Finding the relevant source code files given a bug report is the initial and one of the most important steps for location-based bug assignee recommendation approaches. Therefore, the performance of the bug localization approach used highly impact the performance of a location-based bug assignee recommendation approach. However, bug localization based on a human written report is a hard problem in and of itself, with common accuracy around 30% for predicting the most suspicious source file, e.g., [12].
- **Ignore rich information contained in historical bug reports.** Many location based approaches do not consider the textual information inside previously fixed bug report, which often contain useful information to determine the expertise of a developer.

To summarize, both activity-based and location-based bug assignee recommendation approaches have advantages and disadvantages. In this work, we combine the two to build a unified bug assignee recommendation model that improves on the performance of the previous approaches.

III. A Unified Assignee Recommendation Approach

In this section, we detail our proposed bug assignee recommendation approach. We first introduce the overall framework of our approach (Section III-A). We then introduce the features that we use to capture the degree of match between a developer and a bug report, which include those derived from activity information (Section III-C) and location-based information (Section III-D). We also describe the process of extracting these features.

A. Overall Framework

We apply learning to rank to train a ranking model that uses both activity information and location information as features to identify appropriate developers to address a particular bug report. Learning to rank is a popular machine learning techniques for training a model to solve a ranking problem. It has been widely used in various applications, such as document retrieval [13], [14]. Document retrieval is a task that takes as input a query, and retrieves and ranks documents based on their degrees of match with the query. This problem is similar to our assignee recommendation problem, where a new bug report is the query, and the profiles built from developers' activity information form the documents. To incorporate location information, the query would be enriched with the potential locations where the bug may reside. In this way, we can naturally apply learning to rank to build ranking models for the bug assignee recommendation problem.

TABLE I: Sixteen Activity-Based and Location-Based Features Characterizing a Bug Report-Developer Pair.

Category	ID	Dimension	Description
Activity-Based	ϕ_{1-5}	Bug Report-Code Similarity	Similarity between source files related to the developer and bug report.
	ϕ_{6-10}	Bug Report-Bug Report Similarity	Similarity between previous bug reports related to the developer and bug report.
	ϕ_{11}	Developer Bug Fixing Frequency	How frequently the developer fixes bugs.
	ϕ_{12}	Developer Bug Fixing Recency	How recently the developer fixes bugs.
Location-Based	ϕ_{13-14}	Potential Buggy Code-Related Code Similarity	Similarity between potential buggy files corresponding to the bug report and source files related to the developer.
	ϕ_{15}	Touched Potential Buggy Files	Whether the potential buggy files have been touched by the developer.
	ϕ_{16}	Touched Mentioned Files	Whether classes mentioned explicitly in bug reports have been touched by the developer.

Figure 2 shows the general process of our approach. The recommendation system maintains profiles for all available developers, which we refer to as $D_1, \ldots, D_N$. The main task of this recommendation system is to train a ranking model $f(Br_i, D_j)$ that accurately captures the degree to which a given bug report Br_i matches a given developer j's profile (D_j). To train $f(Br_i, D_j)$, this system requires a set of previously fixed bug reports for which we know the developers to whom they were ultimately assigned. Thus, for a set of M training bug reports $Br_1 \ldots Br_M$ and associated developers $D_1, \ldots, D_N$ the system collects a set of features to represent the degree of match (or similarity) between each bug report and developer. For instance, $d_{1,1}$ represents the similarity between Br_1 and D_1. This information is then used to train the ranking model $f(Br_i, D_j)$ using an off-the-shelf implementation of a learning to rank algorithm. Then, when a new bug report arrives, the trained model calculates the similarity between it and all the potential developers, producing $d_{M+1,1}, d_{M+1,2}, \ldots, d_{M+1,N}$. The output of the whole system for this bug report is a ranked list of developers, where developers at the top of the list have higher similarity scores with the given bug report and are thus more likely to be good choices for addressing the defect.

The ranking model $f(Br_M, D_N)$ is represented as a weighted sum of k features, where each feature $\phi_i(Br_M, D_N)$ captures an element of the similarity between the bug report M and developer N:

$$f(Br_M, D_N) = \sum_{i=1}^{k} w_i * \phi_i(Br_M, D_N)$$

The model parameters w_i are learned from the training set by the learning-to-rank algorithm. The learning-to-rank algorithm employs an optimization procedure that seeks a set of parameters that results in a function that correctly ranks the developer profiles that are known to be assigned to the bug reports in the training set, at the top of the lists for those bug reports.

In the following sub-sections, we introduce 16 features, i.e., $\phi_1(Br_M, D_N) \ldots \phi_{16}(Br_M, D_N)$ that we use to measure the degree of match (or similarity) between a bug report and a developers' profile. These features are derived from developers' bug-fixing activities (see Section III-C) and estimated bug locations in the source code (see Section III-D). Table I summarizes these 16 features.

B. Dataset Collection and Text Pre-processing

In this work, we consider two kinds of resources to build a developer profile that captures expertise: (1) bug reports that have been fixed by the developer, and (2) the corresponding committed source code files. Here, *corresponding files* refer to the files containing code that has been added, modified, or deleted over the course of fixing corresponding bugs. To complete this task, we first collect a set of fixed bug reports and their links to source files committed to a source control system. In this work, we consider three datasets from project Eclipse JDT, Eclipse SWT, and ArgoUML. For Eclipse JDT and SWT, we use the same benchmark dataset provided by Ye et al. [12], where bug reports are already linked with their corresponding bug fixing commits. For project ArgoUML, we apply the heuristic approach proposed by Bachmann and Bernstein [15] to link bug reports with commits. We first scan commit logs to find patterns, such as "issue 180", that could identify bug fixing commits. We then check if the bug reports corresponding to the identifiers exist in the bug tracking system with their status marked as fixed. We also check whether the time the source code files were committed is later than the time the bug report was reported.

After collecting the bug reports and source code files, we extract words appearing in the comments and identifiers of each source code file, and words appearing in the summary and description fields of each bug report. Next, we process the extracted textual information following general textual pre-precessing steps, i.e., tokenization, stop-word removal, and stemming. A token is a string of characters, and includes no delimiters such as spaces, punctuation marks, and so forth. Tokenization is the process of parsing a character stream into a sequence of tokens by splitting the stream at delimiters. Stop words are non-descriptive words carrying little useful information for retrieval tasks. These include verbs such as "is", "am" and "are", pronouns such as "I", "he" and "it", etc. Our stop word list contains 30 stop words, and also common

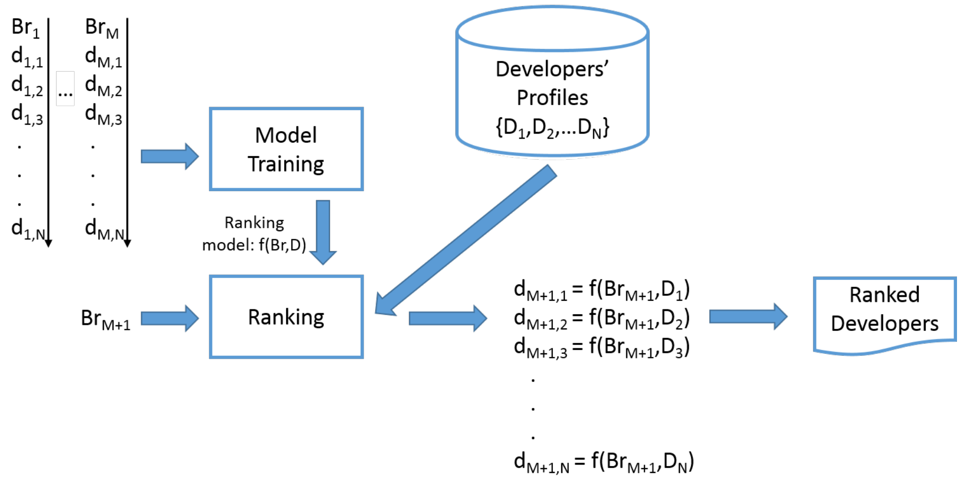


Fig. 2: Overall Ranking Process

abbreviations such as "I'm", "that's", "we'll", etc. Stemming is a technique to normalize words to their ground forms. For example, a stemmer can reduce both "working" and "worked" to "work". We use the Porter stemming algorithm [16] to perform this step.

C. Extraction of Activity-Based Features

This subsection describes features mined from developers' bug fixing activities.

a) Bug Report-Code Similarity: This dimension (ϕ_{1-5}) captures *textual* similarity between a bug report Br and previous files containing source code committed by a developer D to fix prior bugs. We combine the summary and description fields in a bug report into one document per report. We consider a source file that a developer has *touched* (i.e., added, deleted, or modified) as a document. We also create a merged document that contains all files a developer has touched. We consider two metrics to measure document similarity: cosine and BM25 similarity scores.

To compute the cosine similarity (i.e., $\phi_1(Br, D)$) between a bug report Br and a developer D, we first define a function $Cosine(r, s)$ that calculates the cosine similarity between two documents (in our case, a document could be a bug report, a code file or a merged code file) r and s. Function $Cosine$ first transform pre-processed words in document r and s into two vectors of weights. Each word is mapped to an element of the vector. The weight of a word $term$ in a vector corresponding to a document doc is computed as:

$$w_{term,doc} = TF_{term,doc} \times IDF_{term}$$

In the above equation, $TF_{term,doc}$ is the number of times word $term$ appears in doc. $IDF_{term} = log\frac{N}{DF_{term}}$, where DF_{term} is the number of documents that contain word $term$, given a document corpus. TF-IDF (term frequency - inverse document frequency) is a popular way to assign weights in information retrieval [17]. The vector representations of two documents are then compared to compute their cosine similarity as follows:

$$Cosine(r, s) = \frac{\overrightarrow{r} \cdot \overrightarrow{s}}{\|\overrightarrow{r}\|\|\overrightarrow{s}\|} \quad (1)$$

In the above equation, $\overrightarrow{r}$ and $\overrightarrow{s}$ are vector representations of the bug report and the set of patches, $\cdot$ is the dot operation between vectors, $\|\overrightarrow{v}\|$ is the size of vector $\overrightarrow{v}$.

BM25 is another way to compute similarity between documents [18]. Given a query q (e.g., a bug report) and a document s (e.g., a document that contains all source code files touched by developer D to fix prior bugs), $BM25(q, s)$ computes a similarity score as follows:

$$BM25(q, s) = \sum_{i=1}^{n} Idf_{q_i} \cdot \frac{f(q_i, s) \cdot (k_1 + 1)}{f(q_i, s) + k_1 \cdot ((1 - b + b \cdot \frac{|s|}{avgdl}))} \quad (2)$$

In the above equation, q_i is the ith word in the query q, $f(q_i, s)$ is number of times q_i appears in document s, $|s|$ is the length of the document (i.e., number of words in the document), and $avgdl$ is the average document length in the text collection from which documents are drawn (i.e., average number of words in the documents containing touched source code files of different developers). k_1 and b are free parameters. In our experiment, we set k_1 and b as 1.2 and 0.75, as suggested by Manning et al. [17].

Based on these two types of similarity metrics, we define the following five features:

$$\phi_1(Br, D) = max(Cosine(Br, m)|m \in D_{CodeCorpus})$$
$$\phi_2(Br, D) = avg(Cosine(Br, m)|m \in D_{CodeCorpus})$$
$$\phi_3(Br, D) = sum(Cosine(Br, m)|m \in D_{CodeCorpus})$$
$$\phi_4(Br, D) = Cosine(Br, D_{MergedCode})$$
$$\phi_5(Br, D) = BM25(Br, D_{MergedCode})$$

In the above equations, Br is the target bug report. $D_{CodeCorpus}$ is the set of source files touched by developer D to fix previous bugs. For ϕ_{1-3}, we consider single source file (m), as a document, and compute their similarity with the bug report. We then use the maximum value, mean, and sum of these similarity scores as the value of the features. For ϕ_{4-5}, we merge all source files to create a larger document $D_{MergedCode}$ for developer D, and compute its similarity with bug report Br using cosine similarity and BM25.

b) Bug Report-Bug Report Similarity: This dimension (ϕ_{6-10}) captures the textual similarity between a bug report Br and all previous bug reports fixed by a developer D. The underlying idea is that words appearing in the bug reports that have been fixed by a developer might capture the expertise of this developer along different aspects. Similar to ϕ_{1-5}, we consider five kinds of similarity metrics in this dimension, given a bug report Br and a developer D:

$$\phi_6(Br, D) = max(\mathit{Cosine}(Br, m)|m \in D_{BugCorpus})$$

$$\phi_7(Br, D) = avg(\mathit{Cosine}(Br, m)|m \in D_{BugCorpus})$$

$$\phi_8(Br, D) = sum(\mathit{Cosine}(Br, m)|m \in D_{BugCorpus})$$

$$\phi_9(Br, D) = \mathit{Cosine}(Br, D_{MergedBugs})$$

$$\phi_{10}(Br, D) = \mathit{BM25}(Br, D_{MergedBugs})$$

In the above equations, function $Cosine$ and $BM25$ are the same as defined in Equation 1 and 2 respectively. For the feature ϕ_{6-8}, $D_{BugCorpus}$ is the set of bug reports to which developer D was assigned before Br was reported. For the later two features, we merge all documents in $D_{BugCorpus}$ as one document, i.e., $D_{MergedBugs}$, and then compute the similarities between two documents.

c) Developer Bug Fixing Frequency: A developer who has fixed a lot of bugs for a project generally has more expertise on the project compared with other developers. Based on this assumption, we consider the number of bugs that have been fixed by a developer over a period of time (one year, in our experiments) as one of the activity-based features. It is defined as:

$$\phi_{11}(Br, D) = |br_{Oneyear}(Br, D)|$$

In the equation above, $br_{Oneyear}(Br, D)$ is the set of bugs that developer D has fixed within one year prior to the reporting of Br.

d) Developer Bug Fixing Recency: Similar to the intuition captured by ϕ_{11}, we speculate that a developer who has recently fixed bugs might more likely to fix a new bug than another developer who has not fixed any bugs in a long time. Let $br(Br, D)$ be the set of bug reports for which developer D has fixed before bug report Br was reported. Let $last(Br, D)$ be the most recently fixed bug in $br(Br, D)$. Also, for any bug report Br, let $Br.date$ denotes the date when the bug report was created. We then define the bug-fixing recency feature ϕ_{12} to be the inverse of the distance (in months) between Br and $last(Br, D)$:

$$\phi_{12}(Br, D) = (diff^{MTH}(Br.date, last(Br, D).date) + 1)^{-1}$$

In the equation above, $diff^{MTH}(Br.date, last(Br, D).date)$ denotes the difference between the dates Br and $last(Br, D)$ were reported rounded down to the nearest number of months.

D. Extraction of Location-Based Features

a) Potential Buggy Code-Related Code Similarity: To compute these features, we perform two steps: (1) given a bug report, generate a list of source code files that are most likely to be relevant to the bug report using the bug report localization technique proposed by Ye et al. [12], (2) generate features ϕ_{13-15}, to capture degree of relevance between a developer and a bug report by analyzing the potential location of the bug. We consider approach proposed by Ye et al. because it is reported to be the state-of-the-art bug report localization technique so far.

ϕ_{13} and ϕ_{14} correspond to the cosine and BM25 similarity scores between the top-k most likely source code files to contain the bug and a document containing all source code files that are touched by a developer to fix prior bugs. They are mathematically defined below:

$$\phi_{13}(Br, D) = MAX_{C_i \in TopK}(\mathit{Cosine}(C_i, D_{MergedCode}))$$

$$\phi_{14}(Br, D) = AVG_{C_i \in TopK}(\mathit{BM25}(C_i, D_{MergedCode}))$$

In the equation above, $TopK$ refers to a list of top-k files that are most likely to contained the bug described in Br as outputted by Ye et al.'s technique [12]. In the experiment, we set K to 10. $D_{MergedCode}$ is a document that contains all code files touched by D to fix prior bugs.

b) Touched Potential Buggy Files: ϕ_{15} measures whether the developer has touched a file that is potentially buggy when fixing prior bugs. We identify a list of top-K potentially buggy files in a similar way as when we compute ϕ_{13} and ϕ_{14}. In the experiment, we set K to 10 by default.

$$\phi_{15}(Br, D) = \begin{cases} 1, & \text{if developer D has touched } C_i \in \text{TopK} \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

c) Touched Mentioned Files: For some of the reported bugs, developers mention the names of some classes (i.e., source code files) in the description of a bug report [19]. These files are likely to be the buggy ones. Thus, we define another feature as follows:

$$\phi_{16}(Br, D) = |Br.files \bigcap D_{CodeCorpus}|$$

In the equation above, $Br.files$ corresponds to the set of source code files whose names appear in Br and $D_{CodeCorpus}$ corresponds to a set of source code files that are touched by D to fix prior bugs.

IV. Experimental Setup

In this section, we first present the research questions that we have considered in this paper. Next, we describe the datasets that we use in this study, followed by our experimental settings. We then present the measures used to evaluate the approaches, followed by our results. Finally, we also mention some threats to validity.

A. Research Questions

Our core hypothesis is that activity-based and location-based information provide complementary information that can be used to more accurately assign bug reports to developers in a software project. We therefore investigate the following three research questions:

RQ1: Does a bug assignee prediction model that combines activity-based features and location-based features achieve better performance than a model that uses only one type of feature?

RQ2: Does our unified approach outperform existing activity-based or location-based approaches?

RQ3: Which features are the most important to the accuracy of our model?

For RQ1, we compare three results: the results of our unified model using only activity-based features (i.e., ϕ_{1-12}), using only location-based features (i.e., ϕ_{13-16}), and using all features. We use the learning to rank tool named rankSVM [3] provided by Lee and Kuo to train our unified model.

For RQ2, we consider two baselines:

1) **Activity-baseline:** We use the activity-based approach proposed by Anvik et al. that takes words from summary and description of bug reports as features and applies Support Vector Machine (SVM) classifier [3]. Note that this method only returns one label (that is, a single developer) for each bug report.
2) **Location-baseline:** We use the location-based approach proposed by Shokripour et al. [1]. This baseline first uses the weighted sum of common words appearing in bug report and source code file to locate potential files that are related to a bug report. It then recommends a ranked list of assignees based on their recent activity on the potential buggy files.

For the activity-baseline, we use the SVM package in Weka [20] to train SVM classifiers from training data and test it on testing data. For the location-baseline, we write Java code to implement their approach.

For RQ3, we estimate the importance of each features (i.e., ϕ_{1-16}) by considering its corresponding weight w_i (defined in Section III-A), averaged over all training folds.

[3]https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/#large_scale_ranksvm

B. Dataset

We use bug reports from several open source projects: Eclipse JDT,[4] Eclipse SWT,[5] and ArgoUML.[6]. For the first two datasets, we consider the same set of bug reports as Ye et al. in their paper [12]. These bug reports have been linked to commits that fix them. For ArgoUML, we manually download the bug reports and link them to their corresponding bug fixing commits following the heuristics described by Bachmann and Bernstein [15]. Note that we only consider bug reports with status "fixed" for training and testing. Overall, we consider a total of 11,887 bug reports. Table II describes the details of the three datasets.

TABLE II: Datasets: Eclipse JDT, Eclipse SWT, ArgoUML

Project	Time Range	# Bug Reports
JDT	2001-10-10 - 2014-01-14	6,274
SWT	2002-02-19 - 2014-01-17	4,151
ArgoUML	2000-02-01 - 2012-12-13	1,462

C. Experiment Setup and Evaluation Metrics

As described in Section III, our ranking model $f(Br_M, D_N)$ is based on a weighted combination of features that capture domain dependent relationships between a bug report Br_M and a developer D_N. We train the model parameters w_i using the learning-to-rank approach implemented in the rankSVM package [21].

To mitigate the risk of overfitting, we create disjoint training and test data by sorting the bug reports from each benchmark dataset chronologically by reporting timestamp. Note that temporal order matters; for instance, we need to make sure that features are extracted from source code and bug reports that existed prior to the reporting time of a target bug report. For all projects, the sorted bug reports are then split into 10 equally sized folds $\{fold_1, fold_2, \ldots, fold_10\}$, where $fold_1$ contains the oldest bug reports while $fold10$ consists of the most recently reported bugs. The ranking model is trained on $fold_k - fold_{(k+5)}$ and tested on $fold_{(k+6)}$, for all $1 \leq k \leq 5$. In this way, we collect 5 results for each dataset. Since the folds are arranged chronologically, this means that we always train on the previous existing bug reports. For each bug report in a test fold, testing the model means computing the weighted scoring function $f(r, s)$ for each source code file using the learned weights, and ranking all the files in descending order of their scores. We then check if the correct developer (that is, the developer who actually ultimately repaired the bug in question) appears highly ranked in the output list of developers.

Similar to previous work [1], [4], we use Accuracy@K as evaluation metrics. This metric corresponds to the proportion of top-K recommendations that contain the ground truth developer who assigned to the bug report (as recorded

[4]http://www.eclipse.org/jdt
[5]http://www.eclipse.org/swt
[6]http://argouml.tigris.org/

in the bug tracking system). We consider K = 1,2,3,4,5, and 10. For instance, if a assignee recommendation system could successfully capture 30 actual assignees for 100 bug reports at top-1 recommendation, then the value of Accuracy@1 is 0.3.

V. Experiment Results

In this section, we present the results of our experiments in form of answers to research questions 1, 2 and 3. We then discuss threats to validity.

A. *RQ1: Activity-Based Features Vs Location-Based Features Vs All Features.*

In the first research question, we evaluate the efficacy of our unified model for the bug report assignee problem and compare it to models built with each of two types of features alone. The results of our three unified models trained with different sets of features are shown in Table III. From the table, we note that we can achieve an Accuracy@1 of up to 42%, 45%, and 30% on the JDT, SWT, and ArgoUML datasets, respectively. The unified model (with all features) outperforms the other split models in all cases, supporting our claim that there is value in combining activity-based and location-based features in this domain. For Accuracy@1, using all features improves on the results of using activity-based features alone by 12.5%-31.2%, and the results of using location-based features alone by 15.4%-25.0%.

B. *RQ2: Our Unified Model Vs Baselines*

In the second research question, we compare our unified model to state-of-the-art techniques that use activity- vs. location-based features alone. The results of our model and two selected baselines on the three datasets are shown in Table IV. From the table, we note that in most of the cases, our unified model with all features achieves the best results. The activity baseline consistently performs worst in all cases. The location baseline performs better than our model in two cases, i.e., Accuracy@3 on JDT and SWT dataset, and Accuracy@4 on ArgoUML dataset. For Accuracy@1, our unified model can outperform the activity-based baseline by Anvik and Murphy by 50.0%-100.0%, and the location-based baseline by Shokripour et al. by 11.1%-27.0%.

C. *RQ3: Importance of Features*

In our third research question, we analyze our model to identify which features are most helpful to the assignee recommendation process. For each dataset, we consider the average weight of each feature returned by rankSVM tool when building the prediction model. We select the top-5 features for each dataset; they are shown in Table V.

Generally, top-5 features consist of both location-based features and activity-based features. Feature ϕ_{15} (whether a developer has touched a potential buggy file) consistently ranks the first among all features on the three data sets. On the contrary, ϕ_{16}, which refers to how many times a developer has touched a source file directly mentioned in the bug report, does not appear in any top-5 list. Other features, such as ϕ_3 (i.e., sum of bug report-code cosine similarities) and ϕ_{12} (i.e., developer bug fixing recency) also rank highly when considering the three datasets. Comparing bug report-bug report similarity features (ϕ_{6-10}) and bug report-source code similarity features (ϕ_{1-5}), the latter are slightly more important with slightly more features appearing in the top-5 lists. Overall, we note that the models built for all three datasets include features from both types of data included in the approach.

D. *Threats to Validity*

Threats to construct validity relate to the suitability of our evaluation metrics. We consider Accuracy@K as the metric, which is commonly used in previous work in this space [1], [4] and attempts to capture the degree to which our model achieves their stated goals (accurate prediction of which developer should tackle a given report). Similar to previous studies [1], [2], [3], [4], [5], [6], we consider the developer who actually fixed the bug report as ground truth. It is possible that other developers are good alternatives.

Threats to internal validity relates to potential errors in our experiments. We have checked our code, but there might still be errors that we did not notice. Threats to external validity refers to the generalizability of our findings. In our experiments, we consider more than 11,000 bug reports from Eclipse JDT, Eclipse SWT, and ArgoUML. Experiments on these datasets show that our unified model performs better when it combines both activity-based and location-based information. It also outperforms two exiting baselines. To further mitigate these threats to external validity, we plan to experiment with more bug reports from more projects in the future.

VI. Related Work

In this section, we highlight related studies on bug assignee recommendation, and many other tasks related to bug triage process.

A. *Past Studies on Bug Report Assignee Recommendation*

As mentioned in Section II, studies on bug assignee recommendation can be categorized into two groups based on their underlying mechanism: activity-based [2], [3] and location-based approaches [1], [5], [6]. We describe them below. Note that for some work, we only briefly mention them because their details are already introduced in previous sections.

A number of activity-based bug assignee recommendation approaches have been presented in the literature. For example, Cubranic and Murphy collect features from description and summary fields of bug report and build a Naive Bayes classifier for determining the similarity between the expertise of a developer and a new bug report [2]. Later, Anvik and Murphy compare the performance of various machine learning techniques for automatic bug report assignee recommendation task [3], and show that Support Vector Machine (SVM) classifier performs the best among several commonly-used classifiers. Based on this result, we chose SVM as the classifier for our location-based baseline. Most of these approaches

TABLE III: Results of Our Unified Model Trained with Various Features on Eclipse JDT, Eclipse SWT, and ArgoUML Data

Project	Feature	Acc@1	Acc@2	Acc@3	Acc@4	Acc@5	Acc@10
JDT	Activity Only	32%	58%	72%	85%	90%	96%
	Location Only	35%	57%	69%	78%	83%	89%
	All	42%	65%	79%	89%	93%	97%
SWT	Activity Only	40%	62%	77%	89%	92%	97%
	Location Only	39%	60%	73%	81%	86%	92%
	All	45%	66%	80%	90%	94%	98%
ArgoUML	Activity Only	26%	29%	32%	38%	41%	52%
	Location Only	24%	27%	30%	36%	39%	44%
	All	30%	35%	41%	45%	50%	56%

TABLE IV: Results of Our approach and Baselines on Eclipse JDT, Eclipse SWT, ArgoUML

Project	Feature	Acc@1	Acc@2	Acc@3	Acc@4	Acc@5	Acc@10
JDT	Activity Baseline	28%					
	Location Baseline	33%	57%	82%	88%	89%	92%
	All	42%	65%	79%	89%	93%	97%
SWT	Activity Baseline	25%					
	Location Baseline	36%	60%	81%	88%	91%	93%
	All	45%	66%	80%	90%	94%	98%
ArgoUML	Activity Baseline	15%					
	Location Baseline	27%	30%	39%	47%	50%	52%
	All	30%	35%	41%	45%	50%	56%

TABLE V: Top-5 Most Important Features

	JDT	SWT	ArgoUML
Top-1	ϕ_{15}	ϕ_{15}	ϕ_{15}
Top-2	ϕ_{12}	ϕ_3	ϕ_3
Top-3	ϕ_3	ϕ_{12}	ϕ_7
Top-4	ϕ_7	ϕ_{11}	ϕ_{12}
Top-5	ϕ_{14}	ϕ_7	ϕ_1

use term-weighting techniques, such as term frequency-inverse document frequency (tf-idf), to determine the value of word features.

Similarly, a number of location-based bug assignee recommendation approaches have been presented in the literature. For example, Linares-Vasquez et al. used Latent Semantic Indexing (LSI) to first locate potential source files related to a change request and then recommend developers using authorship information in the corresponding source files [5]. Later, Hossen et al. extend Linares-Vasquez et al.'s work by adding more information, i.e., maintainers of relevant source code and change proneness of source code [6]. Shokripour et al. propose a two-phase location-based approach to leverage multiple information sources including identifies and comments in source code files, commit messages, and previous fixed bug reports [1]. In this work, we take their approach as the representative baseline for location-based approaches.

B. Other Studies Related to Bug Triage Process

Beside bug assignee recommendation, many researchers have tried to improve other tasks involved in the bug triage process, e.g., duplicate bug report detection, bug report categorization, bug report prioritization, etc. We describe some of the studies below.

Users of software systems may report bugs that are already exist in the bug tracking system, since bug reporting is an uncoordinated and distributed process. These duplicates need to be manually labeled as such during the bug triage process which takes considerable human effort and time. A number of automated duplicate bug report detection approaches have thus been proposed [22], [23], [24], [25]. Given a new bug report, these approaches return a list of previously reported bugs which are similar to the new report. Runeson et al. extract words from bug report description and summary fields and use cosine, Dice, and Jaccard to measure the similarity of reports [22]. Sun et al. consider not only text in bug reports, but also many other non-textual fields in the bug reports, e.g., product, etc., to capture degree of relevance between two bug reports [24], [25]. They propose a machine learning approach and extend a variant of BM25 to retrieve duplicate reports. Wang et al. enrich textual information from bug reports with execution traces to more accurately detect duplicate bug reports [23]. In this work, we exclude duplicate bug reports as we only consider bug reports that are labeled as "Fixed".

To help bug triagers assign correct categories (e.g., affected component, valid or invalid) to a bug report, Anvik shows that applying an SVM clasifier to bug reports with features extracted from textual information contained in the summary and description fields achieves accuracy between 72% to 92% if top-3 returned categories are considered [26]. Somasundaram and Murphy consider using Latent Dirichlet Allocation (LDA) and Kullback Leibler divergence to represent bug reports and categorize them [27].

Khomh et al. automatically prioritize crash reports for Firefox project based on the frequency and entropy of the crashes [28]. Along similar By contrast, Tian et al. investigate bug reports that are manually submitted by users [29]. They propose a machine learning approach that recommends priority levels based on information available in bug reports by considering many factors from multiple dimensions: temporal, textual, author, related-report, severity, etc.

VII. Conclusion and Future Work

In this paper, we propose a unified model based on learning to rank technique to automatically recommend developers to address particular bug reports. The unified model naturally combines location-based information and activity-based information extracted from historical bug reports and source code for more accurate recommendation. We propose 16 similarity features to capture the similarity between a bug report and a developer profile. We evaluate our unified model on a set of more than 11,000 bug reports from several open source projects: Eclipse JDT, Eclipse SWT and ArgoUML. Our experiments show that combining the two types of features (activity-based and location-based) improves the performance of our unified model as compared to when only one type of features is used. The experiments also show that our unified model performs the best when compared to a location-based baseline by Anvik et al. [3] and an activity-based baseline by Shokripour et al. [1]. Among the 16 features we proposed, we find that feature ϕ_{15} (whether a developer has touched a potential buggy file) is the most important feature in our unified model on all of the three data sets. Feature ϕ_3 (i.e., sum of bug report-code cosine similarities) and ϕ_{12} (i.e., developer bug fixing recency) are the second and third most important features.

In the future, we would like to consider more features that could better represent the degree of relevance between a developer and a bug report. We also plan to compare the execution times of all considered approaches under different settings.

References

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

[2] G. Murphy and D. Cubranic, "Automatic bug triage using text categorization," in *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer, 2004.

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

[4] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 365–375.

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

[6] M. K. Hossen, H. Kagdi, and D. Poshyvanyk, "Amalgamating source code authors, maintainers, and change proneness to triage change requests," in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 130–141.

[7] T. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: better together," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 579–590.

[8] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen, "On the use of stack traces to improve text retrieval-based bug localization," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 151–160.

[9] C. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 181–190.

[10] S. Wang, D. Lo, and J. Lawall, "Compositional vector space models for improved bug localization," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 171–180.

[11] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, p. 391, 1990.

[12] X. Ye, R. C. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 689–699.

[13] T.-Y. Liu, "Learning to rank for information retrieval," *Foundations and Trends in Information Retrieval*, vol. 3, no. 3, pp. 225–331, 2009.

[14] T. Qin, T.-Y. Liu, X.-D. Zhang, D.-S. Wang, W.-Y. Xiong, and H. Li, "Learning to rank relational objects and its application to web search," in *Proceedings of the 17th international conference on World Wide Web*, 2008, pp. 407–416.

[15] A. Bachmann and A. Bernstein, "Software process data quality and characteristics: a historical view on open and closed source projects," in *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, 2009, pp. 119–128.

[16] K. S. Jones, *Readings in information retrieval*. Morgan Kaufmann, 1997.

[17] C. D. Manning, P. Raghavan, and H. Schtze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[18] S. Robertson, H. Zaragoza, and M. Taylor, "Simple bm25 extension to multiple weighted fields," in *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, 2004, pp. 42–49.

[19] P. S. Kochhar, Y. Tian, and D. Lo, "Potential biases in bug localization: Do they matter?" in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 803–814.

[20] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[21] C.-P. Lee and C.-b. Lin, "Large-scale linear ranksvm," *Neural computation*, vol. 26, no. 4, pp. 781–817, 2014.

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

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

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

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

[26] J. Anvik, "Assisting bug report triage through recommendation," 2007.

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

[28] 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 *Reverse Engineering (WCRE), 2011 18th Working Conference on*, 2011, pp. 261–270.

[29] Y. Tian, D. Lo, and C. Sun, "Drone: Predicting priority of reported bugs by multi-factor analysis," in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 200–209.