Applied Soft Computing Journal xxx (xxxx) xxx

Contents lists available at ScienceDirect

Applied Soft Computing Journal

journal homepage: www.elsevier.com/locate/asoc

# Learning to rank developers for bug report assignment

Bader Alkhazi [a], Andrew DiStasi [b], Wajdi Aljedaani [c], Hussein Alrubaye [b], Xin Ye [d], Mohamed Wiem Mkaouer [b,*]

[a] Kuwait University, Kuwait
[b] Rochester Institute of Technology, NY, USA
[c] University of North Texas, TX, USA
[d] California State University, CA, USA

ARTICLE INFO

*Article history:*
Received 18 February 2020
Received in revised form 15 August 2020
Accepted 19 August 2020
Available online xxxx

*Keywords:*
Bug report
Bug assignment
Learning to rank
Software quality
Mining software repositories

ABSTRACT

Bug assignment is a burden for projects receiving many bug reports. To automate the process of assigning bug reports to the appropriate developers, several studies have relied on combining natural language processing and information retrieval techniques to extract two categories of features. One of these categories targets developers who have fixed similar bugs before, and the other determines developers working on source files similar to the description of the bug. Commit messages represent another rich source for profiling developer expertise as the language used in commit messages is closer to that used in bug reports.

In this work, we propose a more enhanced profiling of developers through their commits, which are captured in a new set of features that we combine with features used in previous studies. More precisely, we propose an adaptive ranking approach that takes as input a given bug report and ranks the top developers who are most suitable to fix it. This approach learns from the history of previously fixed bugs to profile developers in terms of their expertise. With respect to a given bug report, the ranking score of each developer is computed as a weighted combination of an array of features encoding domain knowledge, where the weights are trained automatically on previously solved bug reports using a learning-to-rank technique. Our model was evaluated using around 22,000 bug reports, exported from four large scale open-source Java projects. Results show that our model significantly outperformed two recent state-of-the-art methods in recommending the suitable developer to handle a certain bug report. Specifically, the percentage of recommending a developer within the top 5 ranked developers correctly was over 80% for both the Eclipse UI Platform and Birt projects.

© 2020 Elsevier B.V. All rights reserved.

## 1. Introduction

A bug is an issue linked to coding that potentially triggers an abnormal software behavior. When a bug is discovered by software testers, they will open a bug report to initiate the process of correcting it. Bug report assignment is a critical step towards localizing and fixing a bug, as it is the art of matching the open bug report to the appropriate developer who is most likely to process it. Bug assignees typically perform a variety of code reviews and changes to replicate the bug and verify the reported issue to localize it. With the rise in the number of open bug reports, and with the natural increase in the number of teams and developers, matching bug reports to suitable developers becomes challenging, especially because a wrong assignment not only wastes a developer's time but also adds the overhead of re-assigning the bug.

To tackle this challenge, several studies have instigated the design of automated recommendation techniques where relevant bug report information and the history of code changes are mined before assigning the bug report to the appropriate employee [1–3]. Studies analyzing a developer's activities and experience are considered *activity-based*, while studies linking bug reports to a specific location in the code, and so to a potential developer, are considered *location-based*. A recent study by Tian et al. [4] presented a unified model that merges these activity-based and location-based features to benefit from their advantages at the expense of increasing complexity through their combination.

Furthermore, activity-based features heavily rely on profiling developers using their contributions to the project, such as code changes and previously handled bug reports. As the majority of developers' contributions are represented by their code changes, and as code changes represent the largest source of bugs, linking developers' code changes to the open bug reports

* Corresponding author.
*E-mail addresses:* bader.alkhazi@ku.edu.kw (B. Alkhazi), andrewd@rit.edu (A. DiStasi), wajdialjedaani@my.unt.edu (W. Aljedaani), husseina@rit.edu (H. Alrubaye), xye@csusm.edu (X. Ye), mwmvse@rit.edu (M.W. Mkaouer).

https://doi.org/10.1016/j.asoc.2020.106667
1568-4946/© 2020 Elsevier B.V. All rights reserved.

has been proven to be a critical feature for improving the bug assignment process. However, bug report descriptions are written in natural language, while the source code is represented by language instructions. As these two languages differ in context, representation, and expression, this creates a lexical mismatch, hindering the efficiency of existing bug assignment approaches. Therefore, we propose the use of developers' commit messages as a new set of features for the problem of bug assignment. As shown later in Section 4, commit messages are able to bridge the lexical gap as they are written using the same natural language of bug reports, besides containing valuable information that may not be captured by commit code changes.

In this study, we design a model that tackles the problem of assigning the appropriate developers to a given bug report. We first augment our activity-based feature space with a new set of measurements, representing the similarity between the bug report, and the commit messages, to better profile developers. Then we leverage both augmented activity-based features and location-based features to better profile each candidate developer. Since our study includes 22 features, we utilize the ranking variance of the Support Vector Machine (SVM) in order for us to learn how to rank developers according their relevance to a given bug report. To investigate the efficiency of our learning-to-rank model, we evaluate it using a total of 22,416 bug reports, extracted from four popular open-source software systems, namely, Birt, Eclipse UI, JDT, and SWT. Also, to show the impact of the proposed new features, we compared our model with and without them. The SVM rank of our model was also compared with other state-of-the-art ranking algorithms, such as naive aggregation, ordinal regression, and the random ranking as a baseline. Results show the ability of our model to outperform existing ranking models, by achieving an average accuracy of 93% when recommending TOP-10 developers to fix a given bug report.

In summary, our key contributions are as follows:

1. We propose an enhanced activity-based features using information extracted from commit messages. This information can be treated as domain knowledge, which can be used to better profile developers and improve automated assignment of open bug reports. To the best of our knowledge, there is no existing study that used our enriched developer profiling features for the bug assignment problem.
2. We perform a comparative study between our approach and two ranking algorithms that have been known to outperform state-of-the-art algorithms for the problem of bug localization [4,5]. Our key findings show that our approach outperforms existing studies, mainly when the dimensionality of the problem is increased, i.e., the number of candidate developers for potential assignment is high.
3. As we believe in the importance of improving the automation of bug localization and assignment, we encourage the reproducibility and extendibility of our study by providing the dataset and source code of our experiments.[1]

## 2. Background

The bug report life cycle involves various activities, which are outlined in this section. This process begins when the bug report is received by the development team. After that, the team member who is most likely familiar with the faulty source code will be assigned the task. The next step is to localize the bug. This process is performed by the assigned team member(s) who then try to find the ultimate cause and solve the misbehavior problem. Fig. 1 illustrates the life cycle of a bug report.

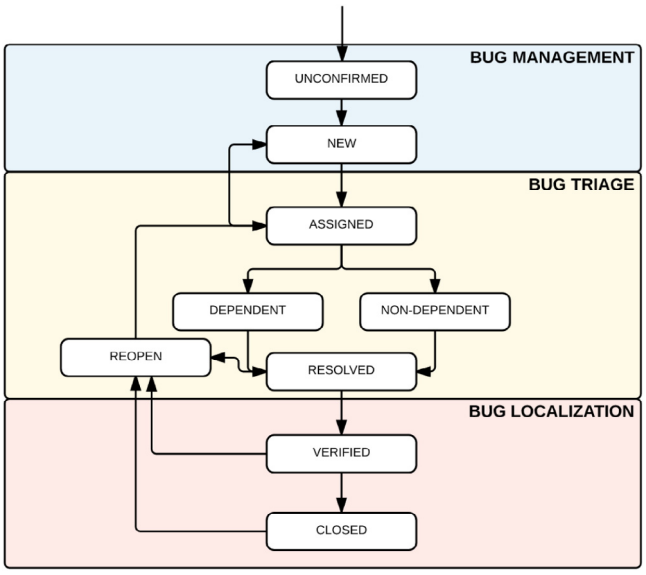


**Fig. 1.** Lifecycle of a bug report.

*Bug:* It is a flaw, fault, or catastrophe in a computer system, resulting in an unanticipated or incorrect outcome or abnormal behavior. Essentially, a bug is something that does not work as designed.

*Bug report:* Is a collection of information regarding the reported issue, it is a way to communicate software flaws to developers to take an action. Usually, the structure of a bug is characterized by the date of discovery, priority, reporter, host software, and description. Bug reports are composed in normal language, unlike source code.

*Bug management:* It is the process of tracking and reporting the progress of a bug from its discovery to its resolution. It is essential to have a repository in software development; it enables developers to report specific issues or bugs. Aside from keeping track of bugs, it also enhances the recommendation of improvements to numerous bug reports. The nature of a certain software project can be improved since the bug repository is publicly available to everyone.

*Bug triage:* It is the process of evaluating, screening, prioritizing, and assigning tracker issues. When a bug is received, an initial screening is performed to determine its validity. Next, bugs will be ranked according to their severity from high to low impact. The bugs that cause the system to be unusable are given a higher rank than issues that cause minor system features to fail [2].

*Bug localization:* It is a task related to software maintenance where developer(s) attempt to interpret the bug report to find the exact location in the source code that is causing the bug.

*Bug tracking systems:* Developers have a common platform where they report various bugs contained in a software product. This facilitates tracking bugs and helps in fixing them efficiently. There are numerous bug tracking systems that are well known, including JIRA, Redmine, Bugzilla, and others.

*Bugzilla:* In this study, we used real-world bug reports from Bugzilla repository as a case study. Bugzilla is one of the most broadly used in both industry and academia [6] since it is the first web-based bug tracking system. Moreover, it is an open-source system where both individuals and enterprises can use freely. The previous studies we compared our results with used these projects which are hosted in the Bugzilla database, thus, to allow for an effective comparison, we used the same project

[1] https://github.com/anonymous-programmers/BugReportAssignment.

Please cite this article as: B. Alkhazi, A. DiStasi, W. Aljedaani et al., Learning to rank developers for bug report assignment, Applied Soft Computing Journal (2020) 106667, https://doi.org/10.1016/j.asoc.2020.106667.

for our study. As shown in Fig. 1, the life cycle of a bug can be assigned one of the following states: *Unconfirmed*, *New*, *Assigned*, *Resolved*, *Reopened*, *Verified*, and *Closed*. Typically, a bug moves from one state to another until it is closed. For instance, new open bugs are marked as *Unconfirmed*, meaning that a bug has been included in the database recently, but the severity of its impact is still unknown. When the developer fixes a bug, they will move it from *Assigned* to *Resolved*. After confirming the work by quality assurance experts, the bug will then be moved to *Verified* state. The full details of the lifecycle can be found in [7].

### 2.1. Information retrieval

Before we introduce the features extracted from the domain knowledge, we will outline the necessary background of the IR techniques used in the extraction process.

#### 2.1.1. Vector space model

The classic vector space model (VSM) can be applied to convert the text into vectors of term weights. The reason to convert the text into vectors is that machine learning algorithms often accept inputs in the form of matrices. In particular, we can take the bug report as a query and source code files as the directory of the documents to be searched from, and then VSM can be used to convert the text of the bug report and source code files into vectors. Those vectors of term weights can be represented as a one-row matrix representing the textual features of each bug report or source code file. In any document $d$ (bug report or source code file), the term weights $w_{t,d}$ of each term $t$ contained in $d$ are calculated as follows:

$$w_{t,d} = \mathrm{TF}_{t,d} \times \mathrm{IDF}_t \tag{1}$$

In the above equation, $TF_{t,d}$ is the number of times term $t$ appears in document $d$. $IDF_t$ is the inverse document frequency = $log\frac{N}{DF_t}$. TF–IDF is one of the widely used methods to convert text into vectors of term weights in IR [8].

#### 2.1.2. Text preprocessing methods and cosine similarity

A similar methodology to the one explained in [9] was applied for text pre-processing. For the bug reports and source code files to be converted into vectors of term weights, first, they need to be preprocessed and cleaned and put into a format that the machine learning algorithms will accept. The first step is to tokenize each body of text by splitting it into its constituent set of words. Then, punctuation and stop words are removed since they do not play any role as features for the learning-to-rank algorithm. Next, all the words are converted to lower case and then stemmed using the Porter Stemmer in the NLTK package. The goal of stemming is to reduce the number of inflectional forms of words appearing in the bug reports or source code files; it will cause words such as "performance" and "performing" to syntactically match one another by reducing them to their base word – "perform". This helps in decreasing the size of the vocabulary space and thus improves the volume of the feature space in the corpus. We used this stemming algorithm not only because it is widely used in previous studies [5,10–14], but also bug reports are written in natural language, and reported by various developers and users to the software, so the text can be generic. So, NLTK is suitable for handling this type of text.

In addition to stemming, another method that is used to reduce the number of inflectional words in bug reports and source code files is Lemmatization. In Lemmatization, the part of speech of a word should first be determined, and the normalization rules will be different for different parts of speech. In this process, the inflections are chopped off based on defined rules, and it relies on some lexical knowledge base to provide the correct basic form of words, while stemming works like a sharp knife, sometimes chopping off too little or too much from the words. For instance, in Lemmatization, the plural word geese changes to goose and the words meanness and meaning remain untouched, while stemming converts both to "mean". Finally, each corpus is transformed into a VSM using the TF–IDF vectorizer in Python's SKlearn package to extract the textual features.

After converting the text in bug report $r$ or source code file $s$ into vectors of term weights, the standard *cosine similarity* method can be used to calculate the textual similarity between them using their vectors of term weights.

$$CosineSim(r, s) = \frac{\overrightarrow{r} \cdot \overrightarrow{s}}{\|\overrightarrow{r}\| \quad \|\overrightarrow{s}\|} \tag{2}$$

We use Cosine similarity because it is also widely used by previous studies [4,15,16], and because the state-of-the-art techniques, that we are comparing with, also use Cosine similarity, so we also use this algorithm to measure the impact of the proposed features on the localization accuracy. Furthermore, Cosine similarity supports measuring how similar the documents are regardless of their size. The cosine of the angle between two vectors is measured and defines if either two vectors refer to nearly the same area. While the two identical documents, despite the size of the text, are far away from Euclidean distances, they are possibly much closer together. The smaller the angle, the more similar the cosine is. Thus, comparing the textual information and documentation in the bug report makes the cosine similarity most relevant to this problem.

Eq. (2) calculates how similar two documents are in the vector space by multiplying the dot product of those two vectors and dividing it by their magnitude. This is a measure of orientation, not magnitude. Here, the magnitude of each vector of term weights in document $d$ is not the only element in calculating the similarity. The angle between the two vectors determines the similarity of the two vectors. For instance, if we have a vector that is quite long and another vector that is shorter, and the angle between those vectors is small, then the cosine similarity tends to ignore the high length and calculates the measurement based on the angle between those two vectors. In practice, if we have a document that contains a word 100 times and another document that contains the same word 25 times, the Euclidean distance between the vector representations of the term weights of those two document will be higher, but the angle will still be small since they are pointing in the same direction, and that determines the similarity score when we compare documents.

## 3. Related work

We discuss in this section the main two threads of related work: (1) Bug assignment approaches, and (2) other studies related to bugs management.

### 3.1. Bug assignment approaches

Several research has been done to find out the best formulation for bug assignment. For instance, Cubranic and Murphy [17] proposed an approach that used naive Bayes classifier to assign bugs to their most appropriate developer. They validated the accuracy of their work using around 16k bug reports from the Eclipse project. Results indicated that their approach was only 30% accurate. Later, the authors in [18] enhanced the aforementioned work by using five term selection methods to boost accuracy and minimize the dimensionality of terms. Moreover, to balance the load between developers, the authors incorporated cost function to distribute tasks evenly.

Please cite this article as: B. Alkhazi, A. DiStasi, W. Aljedaani et al., Learning to rank developers for bug report assignment, Applied Soft Computing Journal (2020) 106667, https://doi.org/10.1016/j.asoc.2020.106667.

The authors in [11] presented a semi-automated approach for bug report assignment. In particular, they used Support Vector Machines (SVM) to present one recommendation for the person in charge of the bug triage process. Results of their experiments on Firefox and the Eclipse projects showed 64% and 57% precision levels, respectively. In the work of Bhattacharya and Neamtiu [19], the researchers used refined classification and intra-fold updates during training to reduce bug tossing paths and improve the accuracy for triaging. The approach was validated on two large projects, Mozilla and Eclipse. Results indicated a prediction accuracy of 83.62% and a reduction in tossing path length for up to 86.31%.

Bugzie was proposed In [20], which is an automatic bug assignment approach based on fuzzy set and cache-based modeling. The basic underlying concept of this approach is to add the developers who are familiar with each technical issue to a fuzzy set. Later, the membership score in the fuzzy set for each developer is continuously updated to reflect he actual contributions of this particular person in regard to recent bug reports. The evaluation of this approach against several existing approaches showed that it achieves better accuracy and lower execution time. In [21], a study was performed to assign developers based on the results of bug's priority, severity, and summary terms analyzed using association rule model. The approach has been validated in multiple projects such as Thunderbird, Bugzilla, and AddOnSDK. In [22], however, a new term-weighting technique was proposed by Shokripour et al. They presented Time–tf–idf, which is an enhanced version of the traditional tf–idf term-weighting technique. The aim of this approach is to accurately identify the expertise of a developer by inspecting the recency of using a particular term by them. Evaluations on three open-source projects showed an upgrade in the mean reciprocal rank and the accuracy of the proposed technique.

Tian et al. [4] introduced the current state-of-the-art approach for generating bug report assignee recommendations. They offered a novel unified learning-to-rank approach that leverages information from both developer activities and bug localization to capture 16 features representing a developer's suitability to fix a bug identified in a bug report. They then evaluated their results on over 11,000 bugs across three open-source projects. Their developer recommendation features are split into two domains: activity-based features and location-based features. They combine these features in a unified learning-to-rank approach that generates a ranked list of the most suited developers to solve a bug report. The authors evaluated their approach against existing activity-based [23], and location-based [24] approaches. They created disjoint test and training sets out of ten equally sized folds of bug reports (sorted temporally, with the oldest reports first) for each project. The ranking model was trained on the five folds prior to a test fold and then evaluated on a test fold to generate a top-k ranking of developer recommendations. Results showed that their unified approach consistently outperformed the existing activity-based and location-based baselines. For top-1 accuracy, their model outperformed the activity-based baseline by 50.0% - 100.0% and location-based baseline by 11.1% - 27.0%. The average weight of each feature returned by the rankSVM tool was used to approximate the relative importance of each feature.

Goyal and Sardana in their study [25] proposed W8Prioritizer and NRFixer. The former is a time-based parameter prioritization approach for bug assignment. While the latter is a prediction model that facilitate forecasting the chances of finding a fix for certain Non-reproducible (NR) bugs.

### 3.2. Other studies related to bugs management

Bug triage process is broader than developer assignment, thus many work focused on other aspects of bug management such as bug report prioritization, duplicate reports, bug localization, etc. We explore some of the studies in this section.

Runeson et al. [10] discussed that each new software release come with several bugs. Moreover, the authors found that in most cases, developers submit bug reports that later are detected as duplicates. Thus, to speed up the bug duplicate detection, they developed a tool using NLP that helps in the detection of duplicates. Using this tool, there are various techniques used in the detection process, such as tokenizing, stemming, and stop word removal. However, it is stated that when the data that to be processed is very large, the operation could be complex. One way to avoid that is to filter out data that are not important and retain only relevant data. In addition, when using an NLP algorithm, the bug reports are ranked based on their similarity so that only the high-ranked bugs are shown. For their case study, they analyzed defect reports at Sony Mobile Communications using two approaches. In the first approach, they chose bug reports that were classified as duplicates in each report using batch runs. In the second approach, they conducted interviews with testers and analysts to evaluate their tool. They concluded that the tool is very useful even though only 40% of duplicates can be detected.

Another approach for locating bugs is presented in [12]. The authors proposed an idea to locate bugs at a code change level. It means that whenever there is a modification within the code, developers can identify those changes that can help find the bug. They proposed a tool named Locus, which provides a better understanding of how bugs can be fixed faster than traditional approaches. Three ranking models are considered, namely, the NL (natural language) model, CE (code entity) model, and boosting model. For the NL, tokens are extracted from bug reports in which only the summary and description sections are considered. Meanwhile in the CE, which refers to the code entity, components such as package names, class names, and methods names are treated as single tokens before preprocessing. For the boosting model, they used the same algorithm used by Google. In this model, the goal is to find the highest suspiciousness score that leads to a file to be buggy. Similarly, authors in [26] pointed out that locating source code that needs to be modified can be tedious. For this reason, they proposed a method named "BugLocator", which is used to locate targeted buggy-files. It is an automated process that reduces waiting time and cost and increases user satisfaction. According to their study, the method outperformed other suggested approaches using a large dataset of bugs from four open-source projects (i.e., Eclipse, AspectJ, SWT, and ZXing).

Other authors [27] proposed a tool named BLUiR (Bug Localization using Structured Information Retrieval), which improves the bug localization techniques. As shown in the results, BLUiR outperformed all other tools. First, it takes a bug report as a query, like other approaches, and then extracts the summary and description, which is tokenized later. Those tokens are converted into queries and are processed for the structured retrieval. Next, the source code files are treated as documents before applying the Abstract Struct Tree on them. Elements such as classes, methods, variables, and comments are tokenized and converted to structured documents that are indexed later before sending them to the structured retrieval.

Ye et al. [28] implemented an IR approach for bug localization that uses adaptive learning to derive weighted features based on the source files, bug history, and API descriptions. Using a learning-to-rank approach, the authors combine features to generate a list of scores indicating the likelihood that the code responsible for the bug exists in the given file. They employ a

Please cite this article as: B. Alkhazi, A. DiStasi, W. Aljedaani et al., Learning to rank developers for bug report assignment, Applied Soft Computing Journal (2020) 106667, https://doi.org/10.1016/j.asoc.2020.106667.

vector space model and cosine similarity for ranking measures of textual similarity. They extract features like those used in this study. They calculate the similarity between a bug report and source code file as well as a bug report and source code file enriched with API specifications for the classes and interfaces. They also implement four metadata features about the bug-fixing history of the project. First, a collaborative filtering score that measures the textual similarity of a bug report to previous bug reports that have resulted in changes to a given file. Next, the similarity of class names mentioned in the bug report to class names in the source code file. Finally, the bug-fixing temporal proximity and frequency for a file, measuring how recently (in months) a file has been changed and how many times within the last year a file has been changed because of bug-fixing activity. They evaluated their model on the dataset that we have selected for use in our study and compared their novel approach against four baselines: a standard VSM method based solely on textual similarity between bug reports and source code, the *Usual Suspects* method that recommends the top-k most frequently fixed files [29], BugLocator ranking tool [5], and BugScout classifier tool [26]. They offer an empirical evaluation of their classifier as compared to these methods, as well as explore issues of future utility, training size and fold count impact, and runtime performance.

Many research works have been focused on either IR or machine learning (ML) techniques for bug localization. Authors in [30], instead, proposed a combined approach to detect the most likely buggy files in projects, which is an advanced IR technique with Deep Neural Network (DNN). Their approach, *DNNLOC*, extracts textual data from bug report and source code pairings, using the same features as [28]. They also added a DNN-based relevancy score that learns to relate terms in a bug report and code or comment text without a reliance on measures of textual similarity, computing how relevant a source code file might be to a bug report. Once a vector of features has been built, the features are fed through a separate multi-layered Deep Neural Network to generate a single combined score that can be used to rank the candidate source files in order of likelihood that they contain the relevant buggy code. This approach was evaluated against the same dataset used by [28] and this paper. Their results indicated a universally higher accuracy at all k-values between 1–20 than all previously identified approaches, including [23, 24,28,29], and [26]. Finally, Safdari et al. [16] used learning-to-ranking strategy to propose a fault localization solution. The study approach utilizes IR and VSM techniques on three types of data source bug report, commit logs and the source code. Their results show that the examined features improve the performance of locating faulty files.

## 4. Feature extraction

This section presents how we extract features from the combination of source files, bug reports, and commit messages.

We cluster these features into the following main categories:

- **Activity-based features.** This category contains all features related to developers' previous experience with resolving bug reports and modified files. For a given bug report, we look for developers who have experience resolving similar bug reports.
- **Location-based features.** This category contains all features related to the location of the bug and relates the most suitable developers who are recently and frequently modified these buggy files, making them more suitable for modifying them again to resolve the given bug report.
- **Profile-based features.** This category contains all features related to the developer's coding activities, and how they correspond to bug report. Since developers regularly document their code changes, it is most likely that the commit message and the bug report may share similarities, and if so, it makes the commit author suitable for dealing with the given bug report.

### 4.1. Activity-based features

This subsection includes features that are calculated based on the bug-fixing activities of developers, including previously touched code and previously solved bug reports.

***Bug report-code similarity***. This section discusses features that measure the similarity between a bug report and code that a developer has interacted with within the past year. For the purposes of this study, we define developer interaction as any action, resulting in that file's inclusion in a commit, including editing, addition, or deletion. For comparison, we aggregate the summary and description into one document for each bug report, *Br*. The first four features are defined as follows:

$$\phi_1(Br, D) = max(Cosine(Br, f)|f \epsilon D_{CodeCorpus})$$
$$\phi_2(Br, D) = avg(Cosine(Br, f)|f \epsilon D_{CodeCorpus})$$
$$\phi_3(Br, D) = sum(Cosine(Br, f)|f \epsilon D_{CodeCorpus})$$
$$\phi_4(Br, D) = Cosine(Br, D_{MergedCode})$$

In the above equations, $Br$ is the summary and description of the bug report. $f$ is each file within $D_{CodeCorpus}$ where $D_{CodeCorpus}$ is the set of source code files in the project touched by the developer, $D$, during any development activity. For $\phi_{1-3}$ we calculate the cosine similarity for each source file and compute the maximum, mean, and sum as feature values, respectively. For $\phi_4$, we merge all touched source code for the developer, $D$, into one document, $D_{MergedCode}$, and compute its cosine similarity with $Br$ as a feature.

***Bug report-API similarity***. Bug Report-API Similarity refers to features that measure the similarity between a bug report and the API documentation of classes and interfaces referenced in the source code that a developer has interacted with within the past year. Typically, most of the text in a bug report is expressed in natural language (e.g., English), whereas most of the content in source code files is expressed in a programming language (e.g., Java). Cosine similarity functions only return non-zero values for tokens that are explicitly present in both documents, meaning that comparing two documents expressed in different forms or languages may not result in a valuable feature. This challenge in comparing bug report text and source code has been termed the *lexical gap*. In this scenario, bug reports and source code are only considered capable of bridging the lexical gap when the source code is annotated with extensive, comprehensive comments or the bug report contains fragments of code matching the source file, such as class names, method names, or a stack trace. However, "in practice, it is often the case that the bug report and a relevant buggy file share very few tokens, if any" [31].

To help bridge the lexical gap, we consider a feature set that measures the similarity between a bug report and the natural language-based API specifications of the classes and interfaces used in the source code that a developer has interacted within the past year. For each source file, we extract references to classes and interfaces from "import" statements and in-line references. Using the project API specification, we extract the natural language descriptions of the referenced classes to create a document for each source file in the project. This technique has been previously employed in [28] to better connect bug reports and source

Please cite this article as: B. Alkhazi, A. DiStasi, W. Aljedaani et al., Learning to rank developers for bug report assignment, Applied Soft Computing Journal (2020) 106667, https://doi.org/10.1016/j.asoc.2020.106667.

files to address the problem of bug localization. These features are defined as follows:

$$\phi_5(Br, D) = max(Cosine(Br, a)|a \epsilon D_{APICorpus})$$
$$\phi_6(Br, D) = avg(Cosine(Br, a)|a \epsilon D_{APICorpus})$$
$$\phi_7(Br, D) = sum(Cosine(Br, a)|a \epsilon D_{APICorpus})$$
$$\phi_8(Br, D) = Cosine(Br, D_{MergedAPI})$$

In the above equations, $Br$ and $D$ are defined as previously described in Bug Report-Code Similarity. $a$ is each set of API documentation for a source file within $D_{APICorpus}$ where $D_{APICorpus}$ is the set of all API documentation for each source file touched by the developer within the past year. To align the range of $\phi_7$ with the other features used in this study, we rescale it using the min–max normalization algorithm. For $\phi_8$, we merge all referenced API documentation for the developer, $D$, into one document, $D_{MergedAPI}$, and compute its cosine similarity with $Br$ as a feature. These features are calculated in the same manner as $\phi_{1-4}$.

***Bug report–bug report similarity***. This section discusses features related to the similarity between a given bug report and any bug reports that the developer has fixed prior to the bug report being filed. These features are calculated in the same way as $\phi_{1-4}$. Again, the summary and description of the bug report are combined into one document for textual similarity computation. These features are defined as follows:

$$\phi_9(Br, D) = max(Cosine(Br, b)|b \epsilon D_{BugCorpus})$$
$$\phi_{10}(Br, D) = avg(Cosine(Br, b)|b \epsilon D_{BugCorpus})$$
$$\phi_{11}(Br, D) = sum(Cosine(Br, b)|b \epsilon D_{BugCorpus})$$
$$\phi_{12}(Br, D) = Cosine(Br, D_{MergedBugs})$$

In the above equations, $Br$ and $D$ are the same as in Bug Report-Code Similarity. $b$ is each bug report within $D_{BugCorpus}$ where $D_{BugCorpus}$ is the set of bug reports marked as "FIXED", which indicates that the developer is the author of the commit that resolved the bug report.

***Bug-fixing metadata***. This section discusses two features related to the bug-fixing history of a project — bug-fixing frequency and bug-fixing recency. Tian et al. assume that a developer who has fixed many bugs is typically knowledgeable about the project and could potentially be more suitable for fixing a bug than a less experienced developer [4]. To quantify this, we include bug-fixing frequency as a feature that enumerates the count of bug reports a developer has completed for a project over the past year. As this feature returns a raw count rather than a ratio or otherwise scaled value, we use a feature scaling approach as implemented in [28] to align this feature with other features. This is defined as follows:

$$\phi_{13}(Br, D) = |br_{Oneyear}(Br, D)|$$

Similarly, a developer who has fixed bugs recently is also typically knowledgeable about the current state of the project and could be more suited for fixing a bug than a developer who has not interacted with the project for some time. To quantify this, we include bug-fixing recency as a feature that enumerates the difference between the dates of the given bug report and the developer's most recent fixed bug report in months (rounded down). This is defined as follows:

$$\phi_{14}(Br, D) = (diff^{MTH}(Br.date, last(Br, D).date) + 1)^{-1}$$

### 4.2. Location-based features

This subsection includes features that are calculated based on the location of the code that likely contains the bug identified in the bug report. To perform this bug localization process, we use the learning-to-rank approach employed in [28].

***Potential buggy code-related code similarity***. To calculate these features, we perform bug localization to generate a list of the 10 source code files that are most likely to contain code relevant to the submitted bug report. We then perform additional cosine similarity calculations to identify the suitability of the developer to work on the likely buggy locations. These functions are defined below:

$$\phi_{15}(Br, D) = MAX_{C_i \epsilon TopK}(Cosine(C_i, D_{MergedCode}))$$
$$\phi_{16}(Br, D) = AVGC_i \epsilon TopK(Cosine(C_i, D_{MergedCode}))$$

In the above equations, $TopK$ is the top-k source code files identified by the bug localization technique to be most likely to contain the code causing the bug described in $Br$. For this research, K is set to 10. We compute the cosine similarity between each file from the localization $C$ and the merged corpus $D_{MergedCode}$ of all code touched by a developer. We return the maximum and average value as features.

***Touched potential buggy files***. This section is a simple indication of whether the developer has touched a file within the top 10 files returned from bug localization used in $\phi_{15}$ and $\phi_{16}$. The feature is defined as follows:

$$\phi_{17}(Br, D) = \begin{cases} 1, & if\ developer\ D\ has\ touched\ C_i \epsilon\ TopK \\ 0, & otherwise \end{cases}$$

***Touched mentioned files***. For some bug reports, developers include the names of some classes or files, either within text or within the form of a stack trace. To capture this, we include the count of files mentioned in the bug report that a developer has previously touched as a feature, defined below:

$$\phi_{18}(Br, D) = |Br.files \cap D_{CodeCorpus}|$$

In the above equation, $Br.files$ is the set of source code file names appearing in the bug report, and $D_{CodeCorpus}$ is the set of source code files touched by the developer, $D$, to fix previous bugs.

### 4.3. Developer profile-based features

This subsection introduces the rationale behind using commit messages as a potential source of information for recommending developer assignment, and then it details how it is represented and calculated as features.

A typical bug report contains a mix of structured information, composed of a natural language text describing the observed anomaly and potential code elements that are involved (*e.g.,* method calls, identifiers, file names, and crash stack traces). As a bug-fixing commit typically contains a similar description, relevant properties can be extracted to characterize how the bug was processed. These properties may not necessarily be reflected in the source code classes and methods being updated by the bug-fixing commit. To provide an illustrative example, we consider the following bug report and its fixing commit in Fig. 2.

As shown in Fig. 2, the developer describes the bug using natural language along with mentioning classes and software versions. Various features correlate this bug report to file changed in its fix commit. However, in this example, the similarity of the source file with the bug report is low, whereas the examination of the commit message log reveals a stronger connection between the bug report and the message's description. For instance, both texts contain similar repeated keywords, such as *extension* and *disabled icons*, increasing their similarity, which enhances the effectiveness of any IR-based approaches [32,33]. Therefore, we propose features capturing the similarity between bug reports and commit messages in the following subsection.

Please cite this article as: B. Alkhazi, A. DiStasi, W. Aljedaani et al., Learning to rank developers for bug report assignment, Applied Soft Computing Journal (2020) 106667, https://doi.org/10.1016/j.asoc.2020.106667.

**Bug Description:** I have declared a toolbar menu contribution command, and the "disabledIcon" property does not seem to be respected. I even tried the org.eclipse.ui.commandImages extension point to verify if the "disabledIcon" from that extension (and not org.eclipse.ui.menus) would be respected, but still no success. What I get on my GUI is the automatically generated version of my disabled icons, which for some images do not look good so I had generated them myself. While running on top of Eclipse 3.7.2 everything looked fine, but now I cannot get it to work.

**Bug-Fix Commit Log:** Bug 384056 - Disabled icons not being used from extension

**Files Changed in the Bug-Fix Commit:**
bundles/org.eclipse.ui.workbench/Eclipse UI/org/eclipse/ui/internal/menus/MenuAdditionCacheEntry.java

**Fig. 2.** Bug Report #384056 from Eclipse UI.

***Bug report-commit log similarity***. If a developer uses a commit message with sufficient description, the message may contain text tokens that relate to the bug report or the code they have fixed. To capture this information, we introduce four additional features. These features aim to measure the textual similarity between a given bug report and the set of previous, non-empty commit messages previously made by the developer. These features are defined as follows:

$$\phi_{19}(Br, D) = max(Cosine(Br, r)|r \epsilon D_{CommitCorpus})$$
$$\phi_{20}(Br, D) = avg(Cosine(Br, r)|r \epsilon D_{CommitCorpus})$$
$$\phi_{21}(Br, D) = sum(Cosine(Br, r)|r \epsilon D_{CommitCorpus})$$
$$\phi_{22}(Br, D) = Cosine(Br, D_{MergedCommits})$$

In the above functions, the cosine function, $Br$, and $D$ are identical to $\phi_{1-8}$. $r$ is each commit message within $D_{CommitCorpus}$ where $D_{CommitCorpus}$ is the set of version control commit messages previously submitted by the developer during development activity. $D_{MergedCommits}$ is a document containing every commit message for that developer. The cosine similarities scores are computed in the same way as $\phi_{1-8}$.

Version history and commit logs have frequently been used in development activity-related data mining, and several previous studies have used them in relating bug reports to developers [34–37]. As stated in Section 4.3, a developer's commit message text can offer a summary of the code written or work performed. As a result, these logs can contain text that may be valuable in matching developers with a bug report. To accommodate this extra dimension of developer suitability, we have introduced four additional cosine similarity features comparing a bug report with documents composed of the developer's commit messages. These features are calculated identically to $\phi_{1-4}$ and $\phi_{5-8}$, using commit logs as the document.

### 4.4. Features summary

Table 1 provides a summary of all the features used in this study.

## 5. Feature combination

This section reviews the three methods we use to combine the features we have extracted from the bug-fixing data. Choosing the appropriate method of feature combination can be as important as choosing the appropriate features themselves. More sophisticated methods of feature combination can be trained to identify the importance of certain features. Accordingly, weighting factor can be applied to emphasize the ones that have been identified as important to the accuracy of the model.

### 5.1. Naive aggregation

To emphasize the need for ranking algorithms, we include a naive aggregation of the features as a baseline for comparison with our other models. Naive aggregation is the most intuitive way to use these features to combine them into one ranking score. Score aggregation was used in earlier studies [5, 26,27] where the number of features is limited, so they are either weighted using trial and error or simply summed to give a final ranking value. In our case, no weights are applied to any of the features. The bug report–developer pairings are then ranked in descending order by this final score. This approach is shown below, where $\phi_i(r, s)$ is each feature score and $k$ is the total number of features:

$$f(Br, D) = \phi(Br, D) = \sum_{i=1}^{k} \phi_i(r, s) \quad (3)$$

### 5.2. Learning-to-rank

We include a learning-to-rank approach for combining features to allow for effective comparison with previous results. This approach was used in [4] to provide their final ranking score. In this approach, a ranking model is trained using historical data that is labeled with the correct developer–bug report pairing. The model uses these values to identify which features are of the greatest importance to the correct bug report–developer parings and calculates corresponding weights for each feature. This approach emphasizes the features that most contribute to a correct ranking and minimizes the ones that do not. Each feature value in the test set is then multiplied by its corresponding weight to generate a final ranking value. The bug report–developer parings are then ranked in descending order by this final score. This approach, as defined by [28], is shown below, where $\phi_i(r, s)$ is each feature score, $w_i$ is that feature's rank assigned by the learning-to-rank model, and $k$ is the total number of features:

$$f(Br, D) = w^T \phi(Br, D) = \sum_{i=1}^{k} w_i * \phi_i(r, s) \quad (4)$$

Just like typical supervised learning algorithms, the training time of SVM Rank depends on the size of the training. The computational complexity is at least quadratic in the number of training instances because of the calculation of kernel matrix. In our case, the dataset contains 22,426, and our various trainings, during experiments were taking less than a minute to generate the rankings.

Please cite this article as: B. Alkhazi, A. DiStasi, W. Aljedaani et al., Learning to rank developers for bug report assignment, Applied Soft Computing Journal (2020) 106667, https://doi.org/10.1016/j.asoc.2020.106667.

**Table 1**
Summary of all used features.

| Feature | Description |
|---|---|
| $\phi_1$ | Maximum Cosine similarity between bug report and the source code touched by developer |
| $\phi_2$ | Average Cosine similarity between the bug report description and the source code modified by the developer |
| $\phi_3$ | Sum of Cosine similarity between bug report and the touched source code by developer |
| $\phi_4$ | Cosine similarity between bug report and the merged code touched by developer |
| $\phi_5$ | Maximum cosine similarity between the bug summary and API documentation |
| $\phi_6$ | Average Cosine similarity between the bug summary and API documentation |
| $\phi_7$ | Sum of Cosine similarity between bug report and API documentation |
| $\phi_8$ | Cosine similarity between bug report and the merged API documentation for the developer |
| $\phi_9$ | Maximum Cosine similarity between the bug report and the developer's previous fixed reports |
| $\phi_{10}$ | Average Cosine similarity between the bug report and the developer's previous fixed reports |
| $\phi_{11}$ | Sum Cosine similarity between the bug report and the developer's previous fixed reports |
| $\phi_{12}$ | Cosine similarity between the bug report and the developer's previous merged fixed reports |
| $\phi_{13}$ | Frequency of bug reports a developer has fixed |
| $\phi_{14}$ | Recency of bug reports fixed by a developer |
| $\phi_{15}$ | Maximum Cosine similarity between each source file from the bug localizer and the merged corpus of all code touched by a developer |
| $\phi_{16}$ | Average Cosine similarity between each source file from the bug localizer and the merged corpus of all code touched by a developer |
| $\phi_{17}$ | Indicator of whether a developer has touched a file within the top 10 files returned by the bug localization |
| $\phi_{18}$ | Number of files mentioned in the bug report that a developer has previously touched |
| $\phi_{19}$ | Maximum Cosine similarity between the bug report and each of the developer's previous commit messages |
| $\phi_{20}$ | Average Cosine similarity between the bug report and each of the developer's previous commit messages |
| $\phi_{21}$ | Sum Cosine similarity between the bug report and each of the developer's previous commit messages |
| $\phi_{22}$ | Cosine similarity between the bug report and all the developer's previous commit messages |

### 5.3. Ordinal regression

To provide an additional dimension of comparison, the features extracted from the dataset were used as input into an ordinal regression model, which was selected due to its suitability in problems involving predicting and ordering ranked values. Predicting an ordinal ranking is a specialized task differentiated from predicting values on a continuous scale by the lack of an intrinsic scale in the numbers representing rank order. For example, predicting students' test scores could be performed by a standard regression model, but a prediction of their class ranking would require an ordinal regression model. The model treats a ranking problem as a series of related classification problems. To address this, the algorithm uses an extended binary classification model for each rank to indicate if the value should be ranked above or below each other rank.

Since it has been used in previous similar studies [30], a two-class boosted decision tree classifier is used as the underlying binary classification algorithm. This is an ensemble learning method where ensuing trees correct for the errors of previous trees before using the entire ensemble of trees to make a joint prediction. We deploy this classifier for this problem as follows:

$$r(x) = 1 + \sum_{k=1}^{K-1} [\![ f(x,k) > 0 ]\!] \tag{5}$$

### 5.4. Random ranker

Similarly to Da Silva et al. [38], we consider the random ranking as one of the baselines to compare against our approach. The rationale behind using this random classifier to hold our approach accountable for providing significantly better results in comparison with a random solution. If the random ranking of developers provides a competitive result, then this problem does not require such complex machine learning models. So, the random ranker is used as a *sanity check* and a baseline.

### 5.5. Feature combination overview

Fig. 3 provides an overview of the process for the learning-to-rank, naive aggregation, and ordinal regression approaches. In this process, a set of developer profiles, composed of previous source code touched, previously resolved bug reports, previous commit messages, and other bug-fixing metadata, are created and mined to generate a set of 18 features that can be used to rank developers in order of suitability for working on a bug report. For the learning-to-rank and ordinal regression, feature scores are extracted from each historical Bug Report–Developer Profile pairing and used to train the ranking model (this does not occur in the naive aggregation approach). When a new bug report enters the system, its feature data are extracted and are then ranked either by the trained ranking model or via naive aggregation. This creates a list of developers ranked in order of suitability to work on that bug report.

## 6. Validation

In this section, we first present our research questions, and then explain our study approach. We then introduce the dataset used and review several corrections and filters applied to the data. Finally, we review the experimental setup and evaluation metrics.

Please cite this article as: B. Alkhazi, A. DiStasi, W. Aljedaani et al., Learning to rank developers for bug report assignment, Applied Soft Computing Journal (2020) 106667, https://doi.org/10.1016/j.asoc.2020.106667.

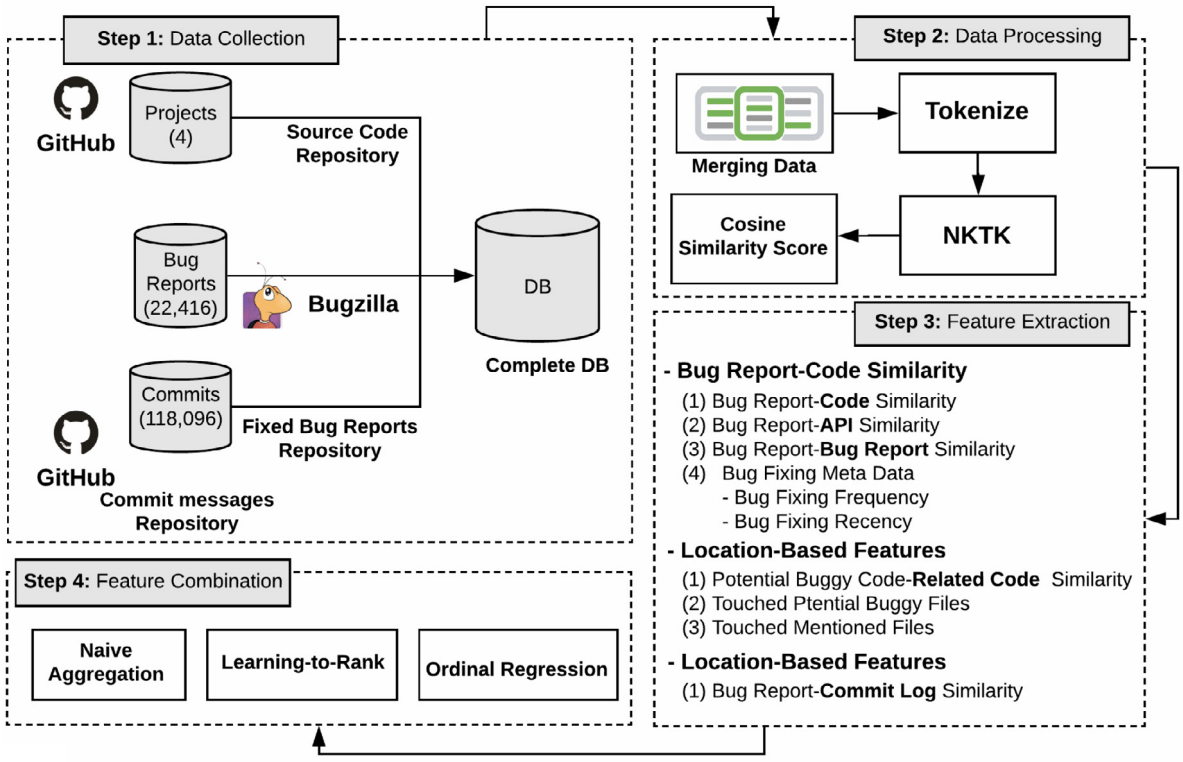


**Fig. 3.** Approach overview.

**Table 2**
Subject projects considered for evaluation.

| Project | # Reports | # Files | # APIs | # Devs | # Commits |
|---|---|---|---|---|---|
| Birt | 4,952 | 6,887 | 971 | 62 | 34,313 |
| Eclipse UI | 6,670 | 3,454 | 1,314 | 122 | 28,621 |
| JDT | 6,528 | 8,184 | 1,329 | 37 | 27,977 |
| SWT | 4,266 | 2,056 | 161 | 31 | 27,185 |
| **Total** | **22,416** | **20,581** | **3,775** | **252** | **118,099** |

### 6.1. Research questions

In attempting to identify if we can more accurately assign bug reports to software developers for a project, we present the following three research questions:

**RQ1:** *Does our model, which considers textual similarity between bug reports and version control commit messages, achieve higher accuracy than baseline models without those features?*

**RQ2:** *Which features are most important to the accuracy of our model?*

We will measure the relative importance of each feature by comparing the weights assigned by the learning-to-rank tool for each feature.

**RQ3:** *Does the size of the training dataset have a significant impact on the accuracy of the classifier for the problem of automated bug assignee recommendations?*

We vary the number of folds used for training for each dataset and examine the impact of smaller training sizes on overall classifier accuracy. We track the accuracy of the L2R+ model using the best performing feature set from RQ1 at k values of 1, 5, 10, and 20 across increasingly small training datasets. With each iteration, the fold containing the oldest data will be discarded until $fold_9$ (the most recent bug-fixing data prior to the test set) is the only fold used for training.

For RQ1, we perform a comparative study between our approach and three different methods of feature combination, representing state-of-the-art studies. The comparative study challenges the ability of existing models to correctly reproduce real-world scenarios of bug report assignments, extracted from our dataset. We compare the following models:

- Our proposed model, learning-to-rank, containing features used in previous studies (*literature features*) (i.e., $\phi_{1-18}$) in addition to our four new features (i.e., $\phi_{19-22}$). We label our approach **L2R+**.
- learning-to-rank, as proposed by [4], containing literature features (i.e., $\phi_{1-18}$). We label it **L2R**. We compare learning-to-rank with the literature feature with the augmented one with our proposed features to show whether they are successful in improving the accuracy of choosing the right developer, for a given bug report.
- Ordinal regression is also used with literature features (i.e., $\phi_{1-18}$). We label it **OR**.
- To see whether the problem of bug assignment can be solved efficiently using a deterministic approach, we used naive aggregation. This helps figuring out whether we need to use a sophisticated learning models for this problem. We aggregate literature features (i.e., $\phi_{1-18}$) and label it **NAGG**.

We design our comparative study using the 10-fold cross validation: To mitigate the risk of overfitting, we create disjoint training and test sets by sorting the bug report data chronologically by when they were reported and divided the dataset into 10 equally sized folds where fold 1 contains the oldest bug reports, and fold 10 contains the most recent ones. Similar to previous studies [28,30], we only consider the most recent previous fold of data when training a model. For this experiment, we consider

Please cite this article as: B. Alkhazi, A. DiStasi, W. Aljedaani et al., Learning to rank developers for bug report assignment, Applied Soft Computing Journal (2020) 106667, https://doi.org/10.1016/j.asoc.2020.106667.

results obtained from training on fold 2 and testing on fold 1, and from training on fold 3 and testing on fold 2. For each bug report, we calculate the feature scores and aggregate them through one of the mentioned scoring functions for each pairing of a developer and the bug report. We then rank these files in descending order based on these results and identify the position in which the correct developer (the developer who submitted the code that resulted in the resolution of the bug report) is ranked. The most recent fold (fold 10) is used as the test set.

We use **Accuracy@K** to evaluate the performance of models. Accuracy@K measures the percentage of bug reports for which we make at least one correct recommendation in the top-k ranked developers. This metric is also used by the previous studies [24,30,39,40]. During the experiment, we vary the K values between 1 and 10, and we report the accuracy of each model for each corresponding value of K.

### 6.2. Study design

Our study approach consists of four main steps. Fig. 3 shows an overview of our approach. The first step is data collection where we first extracted all the necessary data of all the selected projects. The data at this stage includes source code (four projects), bug reports (total of 22,416), and commit messages (total of 118,096). In the second step, we initially merged the bug reports with the relative commit linking them by the bug id. Then, all textual data were pre-processed using standard steps to tokenize, cast to lowercase, remove stop words from, and stem the input. After that we used the Python NLTK standard English stop word list, which includes common pronouns such as "I", "our", and "their", as well as other non-descriptive words such as "are", "these", and "the". Stemming is the process of reducing inflected or derived words to their base form. Then we use the cosine similarity approach to numerically measure the similarity between two text documents.

In the third step, we extract the features from the combination of our crawled data (source code, bug reports, and commit messages). We consider the three main areas that are activity-based feature, location-based feature, and profile-based feature as described in detail in Section 4. Feature combination was the last step of our study; we chose three methods to train the data to identify the importance of certain features. We used naive aggregation, learning-to-rank, and ordinal regression as detailed in Section 5.

### 6.3. Study assumptions

As it is the case in any data mining solution, we have made some assumptions that are important to mention. For each stated assumption, we explain its modeling implication, its rationale, and how to mitigate it in the future.

**Assumption 1.** We assume that every bug report is being handled by one unique developer.

*Rationale.* The reason behind this assumption is that, when we crawl bug reports, we only find one developer assigned to fix it.

*Modeling Implication.* We designed our learning algorithm to consider developers as independent solutions, therefore, no joint ranking is being considered.

*Mitigation.* If a company assigns more than one developer to a given bug report, then our model should be refined to consider such important property. Instead of recommending one developer, the model can recommend *TOP-X* developers, where *X* can be a parameter specified by the user of the model.

**Assumption 2.** We assume that every bug report is being triggered by an error happening in the source files of the software.

*Rationale.* Many software errors can be due to either programming errors, which can be found in the source code, during the debugging process, or they can be due to external factors, linked to the execution environment of the application. Such errors are not handled by our model.

*Modeling Implication.* We designed our learning algorithm to scan all system source files, assuming that the error is due to one of many faulty files. Also, we ruled out bug reports whose status is *INVALID*, to avoid handling errors that are irreproducible.

*Mitigation.* To go beyond source files, future investigation can also include external APIs as another potential source of faults.

**Assumption 3.** We assume that there exists an appropriate developer to fix any given bug report.

*Rationale.* Many software developers rotate across various projects, so there is a chance that the most appropriate developer to handle a given bug report is no longer active or part of the development team.

*Modeling Implication.* As we collect the data for our model, we only considered bug reports for active developers, who are still contributing to the project. So dealing with unavailable developers is out of this work's scope.

*Mitigation.* In the case where the appropriate developer is not available, the model is still able to recommend the second most suitable developer to handle the bug report. Such near optimal solutions are also accepted, since many companies experience employee rotations and turnovers. Therefore, they are flexible with providing tasks to whoever has the best trade-off between experience and availability.

### 6.4. Data collection

In this subsection, we explain the methods used for collecting our dataset. In fact, we started by choosing well-known four open-source projects to post their bug reports into publicly accessible bug trackers. Bug trackers typically use Bugzilla [7] for processing their bug reports. We mainly extracted the bug reports with the status verified fixed, to only consider verified errors that led to changes in the source code, and where properly assigned to a developer to fix them. All duplicate and invalid bug reports were discarded. Additional information about the considered bug reports and their corresponding projects is illustrated in Table 2.

To properly challenge the scalability of our model, we have chosen four open-source projects, belonging to various application domains, implemented by different developers, and hosted in heterogeneous platforms. It is also important to note that we are dealing with heterogeneous set of data, namely natural language descriptions, written in the bug reports, which are compared with a structured language, i.e., source code. Also, we use commit messages which typically contain a mix of short code change descriptions, along with added code elements names or code fragments.

To allow for effective comparison, we used the same dataset used by Lam et al. & Ye et al. and partially overlap with the dataset[2] used by Tian et al. [4,28,30]. Our dataset, along with information about the database structure, source code, feature values, and algorithms used, are made publicly available for use.[3]

[2] Last Access Date: 2020-08-01 Link: http://dx.doi.org/10.6084/m9.figshare.951967.

[3] Last Access Date: 2020-08-01 Link: https://github.com/anonymous-programmers/BugReportAssignment.

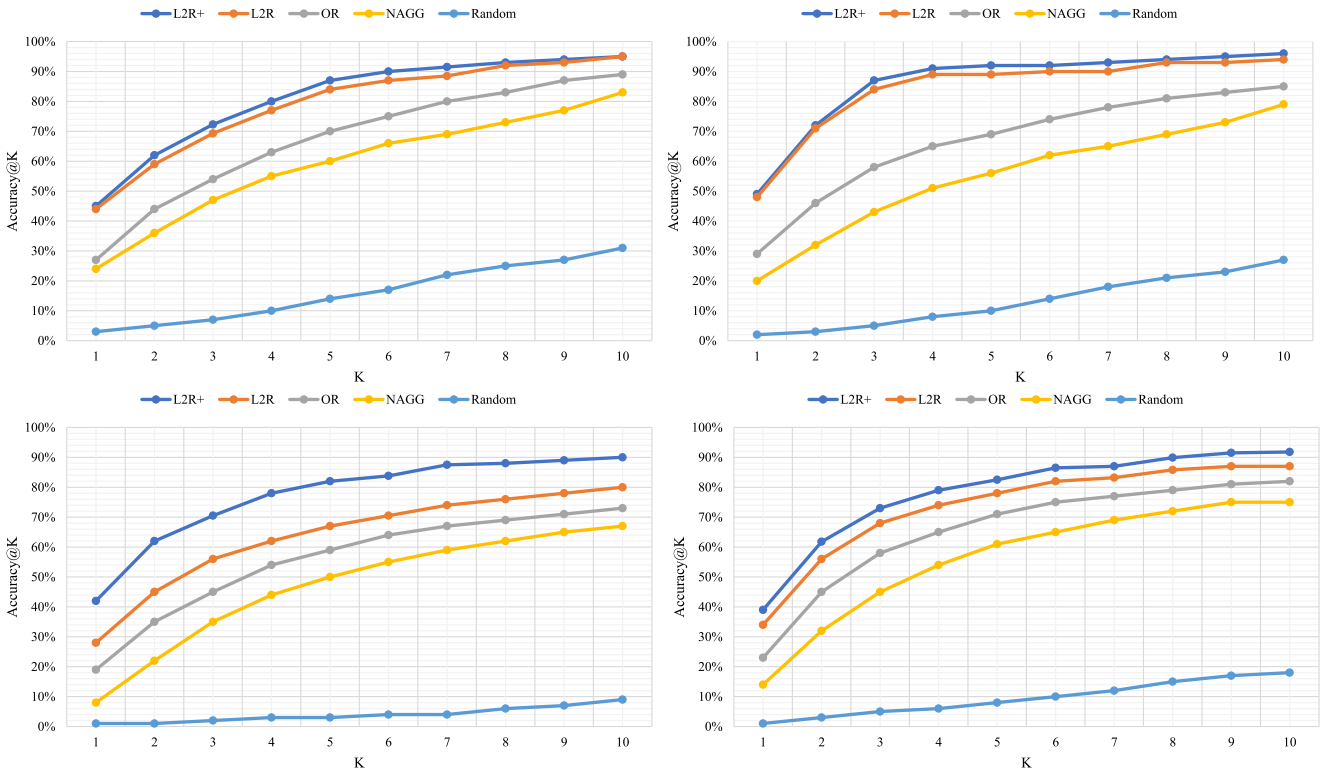


**Fig. 4.** Accuracy comparison between old & new features.

Before using the dataset, we performed a sanity check. Our manual inspection of the data revealed that several developers had entered code under multiple usernames over the course of the project. These multiple usernames occurred due to several reasons, including switching between a username and a display name ("*obesedin*" → "*Oleg Besedin*"), adding a middle name to a display name ("*Markus Kuppe*" → "*Markus Alexander Kuppe*"), and simple typos ("*Chris Goldthorpe*" → "*Chris Goldthorep*"). Each instance of duplicate usernames was manually verified by ensuring that each account shared a common username or email address before correction. These corrections removed ten extraneous developer profiles from the project.

## 7. Results & evaluation

In this section, we review the results from our experiment and discuss them to answer each of our research questions.

### 7.1. RQ1. COmparative study

For the first research question (RQ1), Fig. 4 plots the Accuracy@k results for the four models, with k ranging from 1 to 10, for each project. As shown in Fig. 4, the accuracy of our model L2R+ matches or exceeds the accuracy of the baseline model L2R at all k-values across all projects. Statistical hypothesis testing shows that our approach significantly outperforms the approach used by [4] across all datasets ($p < 0.05$). Furthermore, our approach is particularly effective on the larger Eclipse Platform UI dataset, outperforming the baseline approach by 6%–17% across all k-values, with our accuracy being at least 10% higher for all k values of 1–10. For the JDT and SWT datasets, L2R+ outperforms L2R by up to 4.25%, which is relatively lower than the difference between L2R+ and L2R in Eclipse UI and BIRT. We notice that both JDT and SWT contain 37 and 31 developers, respectively, which increase the chance of picking the right developer for models under comparison, especially when K is higher. However, Eclipse UI and BIRT have 122 and 62 developers, respectively, which expand the search space for the models and so makes the problem harder. However, our approach has shown significantly better performance in these cases as well. Moreover, L2R+ exceeds the accuracy of OR and NAGG at all k-values across all projects, and the difference is statistically significant ($p < 0.05$). The worst results were scored by NAGG. The Naïve aggregation simply treats various features, i.e., rankings equally when determining the final ranking. This experiment empirically demonstrates that few features are more important than others, even if they are all designed for the same goal; that is, recommending the adequate developer, there may be an overlap between features, or they may not be all relevant for a given bug report. Therefore, using a solution that learns from previous bug reports allows the extraction of an ordinal ranking within the features, based on their performance, on individually selecting the right solution (developer), for a given input (developer).

It is important to note that the random ranker has provided the worst results across all solutions, its best performance was scored in the SWT project, where its accuracy@10 has reached up to 30%. This is because the SWT contains 31 developers only, and the probability of selecting the right 10 developers out of them is close to 32%. However, the random ranker has struggled to find the correct developers for the Eclipse UI project. This was due the large number of developers in Eclipse UI project (above 120), hence, the low accuracy@10 score.

In summary, the inclusion of our commit message-related features has significantly improved the accuracy of determining the appropriate developer for a given bug report. We, therefore, believe that these features are also capable of enhancing the performance of the other ranking approaches. Nevertheless, this topic remains a part of our future exploration.

Please cite this article as: B. Alkhazi, A. DiStasi, W. Aljedaani et al., Learning to rank developers for bug report assignment, Applied Soft Computing Journal (2020) 106667, https://doi.org/10.1016/j.asoc.2020.106667.

**Table 3**
Top 5 weighted features for each project.

| Feature weights | | | | | |
|---|---|---|---|---|---|
| Rank | SWT | BIRT | Eclipse UI | JDT | Average |
| 1 | $\phi_{20}$ | $\phi_{20}$ | $\phi_{20}$ | $\phi_{20}$ | $\phi_{20}$ |
| 2 | $\phi_{22}$ | $\phi_{6}$ | $\phi_{22}$ | $\phi_{10}$ | $\phi_{22}$ |
| 3 | $\phi_{10}$ | $\phi_{10}$ | $\phi_{10}$ | $\phi_{8}$ | $\phi_{10}$ |
| 4 | $\phi_{2}$ | $\phi_{3}$ | $\phi_{7}$ | $\phi_{1}$ | $\phi_{1}$ |
| 5 | $\phi_{3}$ | $\phi_{15}$ | $\phi_{1}$ | $\phi_{22}$ | $\phi_{3}$ |

**Table 4**
Description of top 5 features.

| Feature | Description |
|---|---|
| $\phi_1$ | Maximum cosine similarity between bug report and the source code touched by developer |
| $\phi_3$ | Sum of cosine similarity between bug report and the touched source code by developer |
| $\phi_{10}$ | Average cosine similarity between the bug report and the developer's previous fixed reports |
| $\phi_{20}$ | Average cosine similarity between the bug report and each of the developer's previous commit messages |
| $\phi_{22}$ | Cosine similarity between the bug report and all the developer's previous commit messages |

### 7.2. RQ2. FEature importance

To empirically measure the importance of each feature, we can examine the weight assigned to them during the learning-to-rank process. Features with higher weight are highly important for the accuracy of the model, while features with lower or negative weight are less important to the accuracy of the model. To mitigate the risk of an overfitted model, we examined the weights across each individual project, as well as their average. The top 5 weighted features for each project are shown in Table 3. Novel features implemented by this study are shown in bold.

As shown in Table 3, in all projects and for the average weights, two of the four features introduced in our model ($\phi_{20}$&$\phi_{22}$) rank in the top 5 features weights. Specifically, $\phi_{20}$ (the average cosine similarity between the bug report and each of the developer's previous commit messages) has the highest weight across four datasets and the average. $\phi_{22}$ (the cosine similarity between the bug report and all the developer's previous commit messages) is also among the top ranked features and was given the second top place on average. The presence of these features among the top-weighted features across all three folds indicates that our features are both relevant and important to the accuracy of the model. Furthermore, three of our four features were assigned a weight greater than 1, indicating that most of our features are relevant to the accuracy of the model.

The description of top 5 weighted features can be found in Table 4, and an extensive discussion of all features can be found in Section 4.

### 7.3. RQ3. TRaining size

To answer our third research question, we conducted an experiment to evaluate the impact of training data sizes on the accuracy of our classifier. The data from $fold_{10}$ were used as a test set. We initially used all nine remaining folds as a training set. We then reduced the number of folds used in the training set, removing the oldest fold for each trial, until we concluded by just using $fold_9$ (the most recent training data) as the training set.

Fig. 5 shows the accuracy corresponding to the different numbers of folds used in training. Clearly, accuracy does not substantially change when the number of folds changes. Such results can be explained by the temporal locality of bug-fixing activity, where the files that were recently fixed are more likely to require similar fixing again soon. This means that developers assigned to work on these bugs would likely be selected again to work on the new bug. Additionally, this offers the benefit of achieving near-optimal accuracy with a smaller training dataset, reducing the time and memory complexity requirements of training the classifier.

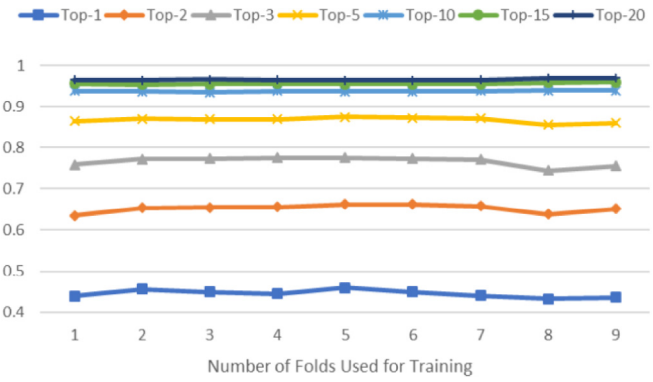


**Fig. 5.** Top-K accuracy with varied training data sizes.

## 8. Future improvements

There are two main areas of improvements to explore for future work: (1) increasing the usefulness of our textual data in feature extraction and (2) exploring the suitability of different algorithms and approaches to aggregating feature scores. To increase the usefulness of our textual data, we identified several dimensions to explore. First, there may be words that, while not present in the standard list of stop words, are used so frequently in bug reports that they lose any value in text similarity comparisons (e.g., "bug" or "fix"). To resolve this issue, we plan to identify the 30 most common words used across the bug report data and include them in the list of stop words removed before cosine similarity calculations. Another way we can improve our cosine similarity calculations is by enriching source code with the API specifications of its classes and interfaces. Given that cosine similarity only returns non-zero scores for documents that explicitly have some token in common, extracting meaningful similarity comparisons between bug reports (written in natural language) and source code (written in programming language) can be difficult. To bridge this lexical gap without a heavy reliance on extensive comments in the source code or the use of programming language in the bug report, we plan to explore the viability of annotating the source code with these APIs or including a separate document containing API annotations as its own feature.

Recent studies have shown that not all bug reports are relevant for a given open bug report, especially if they contain stack traces that are treated as just natural text, so they represent a noise in the search space and a significant computational overhead for the localization and assignment [41]. In this study, we did notice the existence of bug reports containing method invocations and stack traces, and we did not consider any query optimization [42]. For both algorithms under comparison, the processing of features is common. Therefore, query optimization does not impact the comparison; however, we plan to include it in our future experiments to see its impact on the performance of the algorithm.

Please cite this article as: B. Alkhazi, A. DiStasi, W. Aljedaani et al., Learning to rank developers for bug report assignment, Applied Soft Computing Journal (2020) 106667, https://doi.org/10.1016/j.asoc.2020.106667.

### 8.1. Potential applications

The success of our approach in ranking more appropriate developers, can be also be applied to various other potential problems, requiring similar selection of best fitted developer to operate on a specific task [43–45]. Existing studies define developers expertise based on their contribution to the project [46–48]. However, the downside of such approach is the fact of always highly ranking the developer with the largest contributions. Our contribution takes into account the context of the task, and ranks developers who are most likely to be semantically close, and this mitigates the bias of the previous solutions.

## 9. Threats to validity

In this section, we identify several threats to the validity of our study.

**Internal Validity.** It is the level of confidence we have in the cause-and-effect relationship and the factors that might impact our evaluation. Since we are handling informal textual data, we cannot assume that users will follow appropriate grammatical rules. Therefore, we pre-processed, removed stop words, and stemmed data to mitigate this issue. We used the NLTK for text processing, because it was widely used in previous studies, and because it responds to our needs. More tools can be explored to perhaps improve the accuracy of our text preprocessing. Another area of concern is that in some organizations, certain senior developers have the permission to commit changes to sensitive projects. In this case, there is a high chance that those who actually did the work are different from those who committed to the repository. This work, and all other comparable approaches, assume that the developer who made the changes is the same one who committed them. Additionally, bug reports often come from users not developers, these reports may lack technical details or may even provide misleading data. In our study, however, the systems we studied were all used by developers, and the technical vocabulary used in its reports is expected to be closely related to the terms used by developers.

**Construct Validity.** It includes the suitability of our evaluation metrics, the appropriateness of the ordinal regression algorithm, and our assumption of the correctness of the previous work in [30] and [4]. We use Accuracy@K metrics, which have been widely used and validated in a number of related studies [4,20,24, 28,30]. This broad adoption justifies the use of Accuracy@K as our evaluation metric. With regard to the appropriateness of ordinal regression, previous research has shown that ordinal regression is appropriate for ranking problems [49]. Given that this process of bug triage is a ranking problem, ordinal regression is therefore appropriate. Finally, we use the features and methodology of Tian et al. and rely on the dataset and findings of [30], and [4]. We found that the approach, evaluation, and consideration of threats to validity of Tian's approach are sound and comprehensively documented, and we consider their findings valid. The dataset used by Lam et al. has been documented and used in several previous studies [28,30]. Furthermore, we agree with their findings that increasing the size of the training data does not impact the accuracy of the model if the most recent training data are used.

**External Validity.** Our ability to generalize our finding is within the domain of this threat. The main threat in this category is the fact that all datasets used in our evaluation were extracted from open-source projects, all of which were written in Java programming language. Our approach is language agnostic, and the language used to create the approach will not impact the mathematical calculations of the features. The quality of bug reports in our case studies may be different than the quality of other projects. Moreover, commercial projects may have different assignment policies. Thus, we need to further investigate the use of our approach on proprietary projects written in other programming languages in the future.

## 10. Conclusion

This work presents a new set of features to be included in a ranking model to better assign bug reports to the most appropriate developer based on historical bug-fixing data. Our novel contributions introduce the inclusion of four commit message-related features, resulting in a higher accuracy achieved by the model, as well as an empirical evaluation across the two models. Our findings in these studies have indicated that the inclusion of our features contributed to a more accurate model. We conduct a comparative study with state-of-the-art models to empirically validate the appropriateness of our new features. Furthermore, we examine the weights assigned to our features in the learning-to-rank model as compared to other features in the model to verify that our proposed features are relevant to the performance of the model. Results indicate that our model offers higher accuracy than the previous approaches and that our proposed features are relevant to the developer assignment problem.

Additionally, we conduct an evaluation of three different feature combination approaches — naive aggregation, learning-to-rank, and ordinal regression. Evaluation shows that our learning-to-rank approach offers higher accuracy than the other approaches, with a success rate of over 80% in identifying top 5 appropriate developers to handle a bug report. Our experiments have also shown that our machine learning model's training is satisfactory to provide sufficient results.

A future direction in which the study can be further extended is narrowing the focus of the touched code for a developer. Previous studies have considered the entire touched file regardless of the specific contributions the developer made to the file. Narrowing the focus of these features to the method level or to only include the developer's specific contributions could offer a more accurate calculation of the relevancy between a developer and a bug report. Furthermore, the use of different textual similarity calculation methods, such as Word2vec [50], could be explored. Another interesting investigation would be to tackle bug assignment as a multi-objective optimization problem. Multi-objective optimization has shown promising results in solving software engineering problems in general [51–53], and bug localization in particular [44]. In the context of bug assignment, the first objective is to maximize the appropriateness of a given bug report and developers, and the second objective is to minimize the number of recommended developers.

## CRediT authorship contribution statement

**Bader Alkhazi:** Revision, Conceptualization, Methodology, Software. **Andrew DiStasi:** Implementation, Testing, Model Tuning, Writing - original draft. **Wajdi Aljedaani:** Data curation. **Hussein Alrubaye:** Writing - review & editing. **Xin Ye:** Writing - review & editing. **Mohamed Wiem Mkaouer:** Supervision, Writing - review & editing.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## References

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

Please cite this article as: B. Alkhazi, A. DiStasi, W. Aljedaani et al., Learning to rank developers for bug report assignment, Applied Soft Computing Journal (2020) 106667, https://doi.org/10.1016/j.asoc.2020.106667.

[2] G. Jeong, S. Kim, T. Zimmermann, Improving bug triage with bug tossing graphs, in: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM, 2009, pp. 111–120.

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

[4] Y. Tian, D. Wijedasa, D. Lo, C.L. Goues, Learning to rank for bug report assignee recommendation, in: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), 2016, pp. 1–10, http://dx.doi.org/10.1109/ICPC.2016.7503715.

[5] A.T. Nguyen, T.T. Nguyen, J. Al-Kofahi, H.V. Nguyen, T.N. Nguyen, A topic-based approach for narrowing the search space of buggy files from a bug report, in: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 2011, pp. 263–272, http://dx.doi.org/10.1109/ASE.2011.6100062.

[6] R. Malhotra, Empirical Research in Software Engineering: Concepts, Analysis, and Applications, CRC Press, 2016.

[7] Mozilla, Bugzilla tracking system, 1998. https://www.bugzilla.org/. (Accessed 6 August 2020).

[8] C.D. Manning, P. Raghavan, H. Schütze, et al., Introduction to information retrieval, vol. 1 (1), Cambridge university press Cambridge, 2008.

[9] P.S. Kochhar, F. Thung, D. Lo, Automatic fine-grained issue report reclassification, in: Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on, IEEE, 2014, pp. 126–135.

[10] P. Runeson, M. Alexandersson, O. Nyholm, Detection of duplicate defect reports using natural language processing, in: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, 2007, pp. 499–510.

[11] J. Anvik, L. Hiew, G.C. Murphy, Who should fix this bug? in: Proceedings of the 28th International Conference on Software Engineering, ACM, 2006, pp. 361–370.

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

[13] E. AlOmar, M.W. Mkaouer, A. Ouni, Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages, in: 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR), IEEE, 2019, pp. 51–58.

[14] E.A. AlOmar, M.W. Mkaouer, A. Ouni, M. Kessentini, On the impact of refactoring on the relationship between quality attributes and design metrics, in: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2019, pp. 1–11.

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

[16] N. Safdari, H. Alrubaye, W. Aljedaani, B.B. Baez, A. DiStasi, M.W. Mkaouer, Learning to rank faulty source files for dependent bug reports, in: Big Data: Learning, Analytics, and Applications, Vol. 10989, International Society for Optics and Photonics, 2019, p. 109890B.

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

[18] M. Alenezi, K. Magel, S. Banitaan, Efficient bug triaging using text mining, JSW 8 (9) (2013) 2185–2190.

[19] P. Bhattacharya, I. Neamtiu, Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging, in: 2010 IEEE International Conference on Software Maintenance, IEEE, 2010, pp. 1–10.

[20] A. Tamrawi, T.T. Nguyen, J.M. Al-Kofahi, 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, in: ESEC/FSE '11, ACM, New York, NY, USA, 2011, pp. 365–375, http://dx.doi.org/10.1145/2025113.2025163.

[21] M. Sharma, M. Kumari, V. Singh, Bug assignee prediction using association rule mining, in: International Conference on Computational Science and Its Applications, Springer, 2015, pp. 444–457.

[22] R. Shokripour, J. Anvik, Z.M. Kasirun, S. Zamani, A time-based approach to automatic bug report assignment, J. Syst. Softw. 102 (2015) 109–122.

[23] J. Anvik, G.C. Murphy, Reducing the effort of bug report triage: Recommenders for development-oriented decisions, ACM Trans. Softw. Eng. Methodol. 20 (3) (2011) 10:1–10:35, http://dx.doi.org/10.1145/2000791.2000794.

[24] R. Shokripour, J. Anvik, Z.M. Kasirun, S. Zamani, Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation, in: Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on, IEEE, 2013, pp. 2–11.

[25] A. Goyal, N. Sardana, Efficient bug triage in issue tracking systems, in: Proceedings of the Doctoral Consortium At the 13th International Conference on Open Source Systems, 2017, pp. 15–24.

[26] J. Zhou, H. Zhang, D. Lo, Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports, in: 2012 34th International Conference on Software Engineering (ICSE), 2012, pp. 14–24, http://dx.doi.org/10.1109/ICSE.2012.6227210.

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

[28] X. Ye, R. Bunescu, 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, in: FSE 2014, ACM, New York, NY, USA, 2014, pp. 689–699, http://dx.doi.org/10.1145/2635868.2635874.

[29] D. Kim, Y. Tao, S. Kim, A. Zeller, Where should we fix this bug? A two-phase recommendation model, IEEE Trans. Softw. Eng. 39 (11) (2013) 1597–1610, http://dx.doi.org/10.1109/TSE.2013.24.

[30] A.N. Lam, A.T. Nguyen, H.A. Nguyen, T.N. Nguyen, Bug localization with combination of deep learning and information retrieval, in: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), 2017, pp. 218–229, http://dx.doi.org/10.1109/ICPC.2017.24.

[31] X. Ye, R. Bunescu, C. Liu, Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation, IEEE Trans. Softw. Eng. 42 (4) (2016) 379–402, http://dx.doi.org/10.1109/TSE.2015.2479232.

[32] Q. Wang, C. Parnin, A. Orso, Evaluating the usefulness of IR-based fault localization techniques, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ACM, 2015, pp. 1–11.

[33] S. Wang, D. Lo, Amalgam+: Composing rich information sources for accurate bug localization, J. Softw. Evol. Process 28 (10) (2016) 921–942.

[34] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, A. Bernstein, The missing links: Bugs and bug-fix commits, in: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, in: FSE '10, ACM, New York, NY, USA, 2010, pp. 97–106, http://dx.doi.org/10.1145/1882291.1882308.

[35] S. Wang, D. Lo, Version history, similar report, and structure: Putting them together for improved bug localization, in: Proceedings of the 22Nd International Conference on Program Comprehension, in: ICPC 2014, ACM, New York, NY, USA, 2014, pp. 53–63, http://dx.doi.org/10.1145/2597008.2597148.

[36] R. Wu, H. Zhang, S. Kim, S.-C. Cheung, Relink: Recovering links between bugs and changes, in: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, in: ESEC/FSE '11, ACM, New York, NY, USA, 2011, pp. 15–25, http://dx.doi.org/10.1145/2025113.2025120.

[37] V. Sinha, A. Lazar, B. Sharif, Analyzing developer sentiment in commit logs, in: Proceedings of the 13th International Conference on Mining Software Repositories, in: MSR '16, ACM, New York, NY, USA, 2016, pp. 520–523, http://dx.doi.org/10.1145/2901739.2903501.

[38] E. da Silva Maldonado, E. Shihab, N. Tsantalis, Using natural language processing to automatically detect self-admitted technical debt, IEEE Trans. Softw. Eng. 43 (11) (2017) 1044–1062.

[39] H. Alrubaye, M.W. Mkaouer, A. Ouni, On the use of information retrieval to automate the detection of third-party java library migration at the method level, in: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), IEEE, 2019, pp. 347–357.

[40] H. Alrubaye, M.W. Mkaouer, I. Khokhlov, L. Reznik, A. Ouni, J. Mcgoff, Learning to recommend third-party library migration opportunities at the API level, Appl. Soft Comput. 90 (2020) 106140.

[41] M.M. Rahman, C.K. Roy, Improving ir-based bug localization with context-aware query reformulation, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, 2018, pp. 621–632.

[42] M.M. Rahman, C.K. Roy, Improving bug localization with report quality dynamics and query reformulation, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ACM, 2018, pp. 348–349.

[43] A.K. Nassirtoussi, S. Aghabozorgi, T.Y. Wah, D.C.L. Ngo, Text mining for market prediction: A systematic review, Expert Syst. Appl. 41 (16) (2014) 7653–7670.

[44] R. Almhana, W. Mkaouer, M. Kessentini, A. Ouni, Recommending relevant classes for bug reports using multi-objective search, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, 2016, pp. 286–295.

[45] V. Alizadeh, M. Kessentini, W. Mkaouer, M. Ocinneide, A. Ouni, Y. Cai, An interactive and dynamic search-based approach to software refactoring recommendations, IEEE Trans. Softw. Eng. (2018).

[46] D.E. Krutz, N. Munaiah, A. Peruma, M.W. Mkaouer, Who added that permission to my app? an analysis of developer permission changes in open source android apps, in: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), IEEE, 2017, pp. 165–169.

Please cite this article as: B. Alkhazi, A. DiStasi, W. Aljedaani et al., Learning to rank developers for bug report assignment, Applied Soft Computing Journal (2020) 106667, https://doi.org/10.1016/j.asoc.2020.106667.

[47] A. Peruma, M.W. Mkaouer, M.J. Decker, C.D. Newman, Contextualizing rename decisions using refactorings and commit messages, in: 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2019, pp. 74–85.

[48] A. Peruma, C.D. Newman, M.W. Mkaouer, A. Ouni, F. Palomba, An Exploratory Study on the Refactoring of Unit Test Files in Android Applications, in: Conference on Software Engineering Workshops (ICSEW'20), 2020.

[49] B. Gu, V.S. Sheng, K.Y. Tay, W. Romano, S. Li, Incremental support vector learning for ordinal regression, IEEE Trans. Neural Netw. Learn. Syst. 26 (7) (2015) 1403–1416, http://dx.doi.org/10.1109/TNNLS.2014.2342533.

[50] Y. Goldberg, O. Levy, word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method, arXiv preprint arXiv:1402.3722, 2014.

[51] M.W. Mkaouer, M. Kessentini, S. Bechikh, M.Ó. Cinnéide, A robust multi-objective approach for software refactoring under uncertainty, in: International Symposium on Search Based Software Engineering, Springer, 2014, pp. 168–183.

[52] M.W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, M. Ó Cinnéide, High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III, in: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, 2014, pp. 1263–1270.

[53] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, A. Ouni, Many-objective software remodularization using NSGA-III, ACM Trans. Softw. Eng. Methodol. (TOSEM) 24 (3) (2015) 17.

Please cite this article as: B. Alkhazi, A. DiStasi, W. Aljedaani et al., Learning to rank developers for bug report assignment, Applied Soft Computing Journal (2020) 106667, https://doi.org/10.1016/j.asoc.2020.106667.