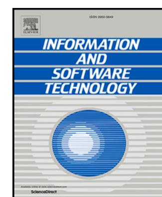
Information and Software Technology 149 (2022) 106956

Contents lists available at ScienceDirect

Information and Software Technology

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



Cleaning ground truth data in software task assignment

K. Ayberk Tecimer [a,*], Eray Tüzün [b,*], Cansu Moran [b], Hakan Erdogmus [c]

[a] Department of Informatics, Technical University of Munich, Munich, Germany
[b] Department of Computer Engineering, Bilkent University, Ankara, Turkey
[c] Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, USA

ARTICLE INFO

Keywords:
Task assignment
Code reviewer recommendation
Bug assignment
Ground truth
Labeling bias elimination
Systematic labeling bias
Data cleaning

ABSTRACT

Context: In the context of collaborative software development, there are many application areas of task assignment such as assigning a developer to fix a bug, or assigning a code reviewer to a pull request. Most task assignment techniques in the literature build and evaluate their models based on datasets collected from real projects. The techniques invariably presume that these datasets reliably represent the "ground truth". In a project dataset used to build an automated task assignment system, the recommended assignee for the task is usually assumed to be the best assignee for that task. However, in practice, the task assignee may not be the best possible task assignee, or even a sufficiently qualified one.
Objective: We aim to clean up the ground truth by removing the samples that are potentially problematic or suspect with the assumption that removing such samples would reduce any systematic labeling bias in the dataset and lead to performance improvements.
Method: We devised a debiasing method to detect potentially problematic samples in task assignment datasets. We then evaluated the method's impact on the performance of seven task assignment techniques by comparing the Mean Reciprocal Rank (MRR) scores before and after debiasing. We used two different task assignment applications for this purpose: Code Reviewer Recommendation (CRR) and Bug Assignment (BA).
Results: In the CRR application, we achieved an average MRR improvement of 18.17% for the three learning-based techniques tested on two datasets. No significant improvements were observed for the two optimization-based techniques tested on the same datasets. In the BA application, we achieved a similar average MRR improvement of 18.40% for the two learning-based techniques tested on four different datasets.
Conclusion: Debiasing the ground truth data by removing suspect samples can help improve the performance of learning-based techniques in software task assignment applications.

1. Introduction

Task assignment in software engineering is concerned with assigning a developer(s) to a development-related task such that the assigned person is capable of completing the task effectively, expediently, and with acceptable quality [1]. Several studies in the literature proposed techniques to automate assignments tasks [2–5]. The techniques develop models based on historical data, which need to be reliable and accurate enough for the models to perform optimally in practice. In previous work, Tuzun et al. [6] investigated problems that can plague historical data in this and other contexts in software engineering, and proposed strategies that can be applied to improve both data quality and model performance. These problems occur when the proposed techniques blindly rely on the data as absolute ground truth: however historical data that involve human decisions come with biases that are baked into those decisions. Two instances of task assignment discussed in Tuzun et al. are CRR and BA. This paper presents two applications, i.e., CRR and BA, that demonstrate how cleaning up the ground truth to eliminate suboptimal decisions from the historical data can improve the performance of models that rely on the historical data.

The code review process is an important step in the software development lifecycle. Effective code reviews increase internal quality and reduce defect rates [7]. For effective code reviews, reviewers should be selected carefully. According to Google's best practices for code reviews [8], "the best reviewer is the person who will be able to give you the most thorough and correct review for the piece of code you are writing". Several CRR techniques exist in the literature [9–17]. These CRR techniques use different strategies, but they invariably either build or evaluate their models based on datasets gathered from industrial or

* Corresponding authors.
E-mail addresses: ayberk.tecimer@tum.de (K.A. Tecimer), eraytuzun@cs.bilkent.edu.tr (E. Tüzün), cansu.moran@ug.bilkent.edu.tr (C. Moran), hakane@andrew.cmu.edu (H. Erdogmus).

https://doi.org/10.1016/j.infsof.2022.106956
Received 15 November 2021; Received in revised form 14 May 2022; Accepted 17 May 2022
Available online 25 May 2022
0950-5849/© 2022 Elsevier B.V. All rights reserved.

open-source projects. Hence they rely on the datasets accurately capturing the ground truth regarding past reviewer selections. The models assume that a code reviewer assigned to a review task, often determined in a *pull request* (PR), in a dataset is the best possible reviewer for the task. However, in practice, the selected code reviewer may not be the most qualified, or even sufficiently qualified to review the submitted PR [18]. In several cases, reviewer assignments can be based on non-technical factors, which may invalidate the central assumption that the models are built on [18]. This situation was described in a study on code reviewer practices at Microsoft [19], where reviewers are assigned to PRs according to their availability and social relationship with the person who makes the reviewer assignments. According to Dogan et al. [18], availability is an important factor for reviewer assignments and is frequently substituted for technical or competency factors. Consequently, recommendation labels in datasets that originate from real practice may be suboptimal, and can negatively affect the accuracy and reliability of the CRR techniques that rely on them.

BA is another important task in managing software projects. Many open-source projects contain open bug repositories, where both developers and users report the problems they encounter. Additionally, they can recommend changes and enhancements to improve the overall quality of the software [3]. As in CRR, for each reported bug, often an assigner selects an appropriate developer who has the expertise to perform the fix. The assignment can be manual or automated. In automated BA, the assignment is decided by heuristics or trained models, using and combining text categorization [20], machine learning [3,21], artifact/dependency analysis [22,23]. Regardless of the technique used, historical bug assignment data that originate from proprietary or open-source projects involving human decision makers play an important role. Thus, as in the case of CRR, the data is susceptible to biases resulting in suboptimal assignments where an assigned developer might not have been the most qualified, or even sufficiently qualified, for the task.

In machine learning, the kind of labeling error that exists in CRR and BA datasets is generally referred to as *systematic labeling bias*. Supervised learning techniques require labels in the training samples. These labels indicate real/actual classes of interest in past data so that models can be built to predict the classes of new data. For instance, in order to distinguish between apples and oranges, an actual label (i.e., apple or orange) for each training sample is required. Ground truth refers to these labels indicating the actual class of the training samples. In more complex tasks of pattern recognition, such as the classification of code review tasks according to who should review them, 100% "correct" labels may not be present in the training samples due to several factors, including subjective ones, that need to be considered. The notion of a "correct" label is not well-defined in such a context. Although the labels are not perfect, they are still considered as the ground truth. When the amount of problematic labels in the ground truth is relatively small or inconsistencies in the class labels are negligible, associated samples can be treated as normal noise. But in some cases, the ground truth may include more pervasive problems due to basic/naïve assumptions in the labeling process [24] or intrinsic properties of the observed data [25] that can prevent the models from converging, learning generalizable patterns, or, as in the CRR and BA cases, being as effective in real practice as they could be. These cases are said to have systematic labeling bias. To the best of our knowledge, automatic cleaning ground truth data in the software task assignment problem has not been investigated by any other study in the literature.

With the goal of preventing labeling problems of the kind described above for task assignment automation in software engineering, we formulate two research questions:

RQ1: *How can we eliminate systematic labeling bias in task assignment ground truth data?*

RQ2: *How does systematic labeling bias elimination in the ground truth data affect the performance of task assignment techniques that rely on the data?*

For RQ1, we explore possible solutions and introduce a new approach to detect and eliminate potentially "incorrect" assignee labels in CRR and BA datasets. For RQ2, we measure the effects of our proposed approach by comparing *before* and *after* accuracy rates of five CRR techniques and two BA techniques. The five CRR techniques are Naïve Bayes, k Nearest Neighbor (k-NN), Decision Tree, RSTrace [26] and Profile based [27]. The two BA techniques are deep-learning based bug triage — Deep Triage [28] and Convolutional Neural Network (CNN) based word representation — CNN Triage [29].

Section 2 provides a summary of relevant previous work on CRR approaches, BA approaches, cognitive biases in software engineering, and ground truth problems in software engineering. Section 3 defines success criteria for CRR and BA for correct assignments. Furthermore, Section 3 introduces our debiasing (data cleaning) approach. Section 4 describes our experiments underlying the two applications, introducing first the datasets, preprocessing steps, and experimental setups, and then presenting the results. Section 5 answers the research questions and discusses the limitations of our work. Finally, Section 6 summarizes the contributions and discusses future work.

2. Background and related work

In the following, we provide a summary of CRR and BA techniques, related work on cognitive biases and ground truth problems in software engineering.

2.1. Code reviewer recommendation techniques

The CRR techniques discussed in the literature fall under mainly two categories: optimization-based approaches and learning-based approaches. We mention representative works below to illustrate the diversity of the approaches. For a more detailed overview of CRR techniques, please refer to Cetin et al. [30].

Optimization-based approaches. Balachandran [9] proposed a heuristic that analyzes the change history to find suitable reviewers. They reach 60%–92% recommendation accuracy, which is better than a comparable CRR approach based on file change history. Lee et al. [10] proposed a graph-based technique to find reviewers in open-source projects. Their method achieves an average recall of 0.84 for Top-5 predictions and a recall of 0.94 for Top-10 predictions. A technique based on analyzing file-path similarity was developed by Thongtanunam et al. [11]: later Xia et al. [12] extended this technique with text mining to leverage additional information in recommendations. Ouni et al. [13] proposed a search-based genetic algorithm to identify most appropriate peer reviewers for their code changes. The authors evaluate their approach on three different open-source projects (e.g., Qt, OpenStack, and Android). Their experiments show that their genetic algorithm recommends code reviewers with up to 59% of precision and 74% of recall. Zanjani et al. [14] presented a technique focusing on previous review quality of candidate reviewers. They argue that providing specific information (e.g., quantification of review comments and their recency) significantly improves CRR approaches. Sulun et al. [26] proposed a graph-based technique using traceability relations between PRs, source code files, and bugs to recommend code reviewers for a given PR. Rebai et al. [31] proposed applying a multi-objective search algorithm, NSGA-II, to obtain a CRR technique that finds a balance between the objectives of expertise, availability and history of collaborations during recommendation.

Learning-based approaches. Jiang et al. [16] proposed a technique that builds a model with Support Vector Machines (SVM) to make reviewer recommendations in open-source projects. The authors evaluate their technique on 18,651 pull requests of five popular projects on GitHub. They indicate that their technique achieves an accuracy between 72.9% and 93.5% for Top-3 recommendation. Xia et al. [17] presented a hybrid approach in which they combined latent-factor models and neighborhood methods. Their results demonstrate that the

proposed approach performs better than comparable methods for all Top-k recommendations. Strand et al. [32] developed a CRR technique that considers workload of possible reviewers and attempts to suggest a reviewer such that the review load is balanced.

All the CRR techniques described above evaluate their models based on datasets collected from real projects, where a reviewer assigned to a code review task is assumed to be the right, or best, reviewer for the job. However in practice, this assumption is often violated, making the ground truth suspect. Some authors [13,14,17,33] acknowledge this problem explicitly as a limitation. For instance, Ouni et al. [13] discuss that reviewers who are assigned to a PR may not do the job well for various reasons (such as workload or availability), or the review may end up being poor quality because the assignment was mainly determined by social factors rather than competence. Lipcak and Rossi [33] state that evaluating CRR techniques with Top-k-style criteria may not be accurate as it is not guaranteed that the actual reviewers in the test data were the best candidates (or even sufficiently qualified) for the tasks for which they were selected.

2.2. Bug assignment techniques

BA entails assigning a reported bug to an appropriate developer capable of fixing it. This can be done by a human manually or with the help of an automated technique. Similar to CRR approaches, BA techniques can be optimization-based or learning-based. We mention some representative works in each category below.

Optimization-based approaches: Zhang et al. [34] proposed a combination of information retrieval and machine learning techniques to identify historical bug reports similar to a new bug. They utilized the REP algorithm and k-NN classification for searching similar bugs and used extracted features from these bugs to develop severity prediction and bug assignee recommendation heuristics. They evaluated their approach on five open-source projects, including GNU Compiler Collection (GCC), OpenOffice, Eclipse, Netbeans, and Mozilla, and found that their approach can outperform other BA techniques such as DRETOM, DREX, and DevRec [34].

Xia et al. [35] introduced a specialized optimization-based topic modeling technique named multi-feature topic model (MTM) to map bug reports to specific topics and an incremental learning method named TopicMiner to find an appropriate developer based on the developer's affinity to the identified topics of the new bug. They evaluated their approach on five projects: GCC, OpenOffice, Mozilla, Netbeans, and Eclipse. Their experiments show that TopicMiner can achieve Top-1 and Top-5 prediction accuracies within the ranges 48%–68% and 76%–90%, respectively.

Sun et al. [22] presented a novel bug fixer recommendation method named BugFixer, which constructs a Developer-Component-Bug (DCB) network to model the relationships between the developers, source code components, and bug reports. Their experiments show that BugFixer can outperform competing methods for large projects and achieve comparable performance for small projects. Naguib et al. [23] introduced a BA recommendation heuristic that profiles the developers according to their previous activities in the bug tracking repository. Each profile is used to identify the most suitable assignee for a particular bug according to their roles, expertise, and involvements in the project. Their approach was evaluated on three different projects. Their experiments show that their approach can achieve an average hit ratio of 88%.

Learning-based approaches: Anvik et al. [3] proposed a semi-automated approach that utilizes SVM to recommend an appropriate developer to address a given bug. They evaluated their approach on Eclipse and Firefox projects and achieved 57% and 64% precision levels, respectively. Murphy et al. [20] presented a supervised Bayesian learning approach to determine the appropriate developer for a bug based on features extracted from the bug description. The authors evaluated their approach on the Eclipse dataset and achieved 30% accuracy. Anvik et al. [23] introduced a recommender system for BA with two different deep learning classifiers. The authors evaluated their Convolutional and Recurrent Neural Network classifiers on Netbeans, Eclipse, and Mozilla projects. Jonsson et al. [21] studied an ensemble of machine learning algorithms, including Naïve Bayes, SVM, and Decision Trees, to automate the BA process. They evaluated their classifiers on five industrial projects. Lee et al. [28] built an automatic bug assigner with CNN and evaluated their model on both open-source and industrial projects. Their experiments showed that their CNN approach could achieve 61% accuracy on a dataset where human assigners achieved 41% accuracy. Mani et al. [36] suggested an Attention-Based Deep Bidirectional Recurrent Neural Network (DBRNN-A) with a softmax classifier. They hypothesized that including the bug description as an input in addition to the bug title would increase performance. The authors evaluated their models on three different open-source projects. Their model outperformed others that adopted the bag-of-words approach in terms of Top-10 average accuracy. Zaidi et al. [29] presented a CNN where the bug descriptions were embedded using three different word embedding techniques. Bhattacharya et al. [37] suggested a set of machine learning techniques and a probabilistic, graph-based model with which they achieved up to 86.09% prediction accuracy.

As with the CRR techniques, the BA techniques mentioned above evaluate their models based on datasets collected from real projects, where past assignment decisions were assumed to be optimal. Again, as in the case of CRR, in practice, suboptimal bug assignments due to convenience or other cognitive biases can pollute the ground truth, limiting the performance of the models.

Other researchers have proposed automated techniques for cleaning ground truth data for software engineering problems outside of task assignment. Yang et al. [38] proposed an automated dataset purification technique to filter code changes irrelevant to a bug. They argued that, when training automated program repair techniques, relying on manual patches as the ground truth could impact the assessment of test-suite-adequate patches due to the likelihood of manual patches having been mixed with other code changes irrelevant to the bug. Therefore, they suggested an automated approach to isolate the segments of the code where the actual bug was fixed from the irrelevant code segments included in the patch. Their work is similar to our approach in that it also treats human-produced data as being prone to noise. However, the context in which the authors apply their approach in filtering human-produced patches differs from ours: cleaning ground truth data in software task assignment. Moreover, they deduce the relevancy of a line of code to the given bug through delta debugging. In contrast, we identify an objective success measure customized to the type of task assignment.

2.3. Cognitive biases in software engineering

A cognitive bias is a type of systematic error in decision making that causes suboptimal outcomes. The error occurs due to established beliefs and misguided intuition. Our past work [39] on improving the performance of CRR techniques addresses systematic labeling errors that in great part stem from such biases.

Cognitive biases are pervasive in software engineering. Mohanani et al. [40] provided many examples of cognitive biases in software engineering in their literature review and advocated their mitigation as central to improving the quality of the decision making. Ralph [41] attributed the persistence of high software project failure rates despite important advances in software technologies and processes to cognitive biases. Stacy and Macmillian [42] stated that cognitive biases have an adverse effect on software developers' thought processes. They advised empirical investigation rather than relying on intuition, and seeking disconfirmatory information to reduce cognitive biases. Smith and Bahill [43] similarly argued that cognitive biases disturb rational decision making in systems engineering through a mechanism called

attribute substitution: the substitution of a convenience factor for an objectively important factor in a decision. They suggested that raising the awareness of this mechanism among engineers could help alleviate its adverse effects in engineering decisions. Attribute substitution is particularly relevant to our work since it is often a primary source of failed labels in CRR and BA datasets caused by past task assignments, where convenience and social factors are substituted for factors that are directly related to the suitability of an assignee for the specific task.

2.4. Ground truth problems in software engineering

Outside BA and CRR [18], ground truth problems exist in data-analytics solutions that support other common prediction and recommendation tasks in software engineering.

Bird et al. [44] studied bug-fix datasets and found strong evidence of systematic bias due to mislabeling of bug fixes in version histories. The performance of a defect prediction model that they tested was adversely affected when the model was built from biased data.

Nguyen et al. [45] examined tagging and linkage biases in IBM Jazz software. Tagging bias results from treating all logged issue as bugs (some of which can represent other coding tasks, decisions, or enhancements). A linkage bias occurs when there is no traceability connection between a bug-fix PR and the corresponding bug report. The authors found linkage and tagging biases even in datasets previously thought to be near accurate.

Herzig et al. [46] analyzed tangled changes in defect prediction. A tangled change is caused by bundling multiple unrelated changes in a single commit. Tangled changes introduce noise to the data, where as much as 17% of all source files could be incorrectly associated with bug reports due to such tangling. The authors state this can negatively impact defect prediction models.

Ahluwalia et al. [47] investigated biases in datasets that were used to build defect prediction models. The authors stated that the bugs were usually discovered after several releases, and therefore may have still been dormant in the snapshot taken to build the models. According to the study, dormant bugs exist in up to 20% of existing releases in a dataset, distorting the ground truth by causing defective code to be mislabeled as defect-free. The authors analyzed 282 releases from six open source projects and demonstrated the existence of ground truth problem in bug datasets, however they did not propose a solution for debiasing the data. Chen et al. [48] also analyzed dormant bugs, using the Apache codebase. They observed a higher dormant bug rate of 33% than that reported by Ahluwalia et al. Both studies demonstrate that the existence of dormant bugs could potentially affect the reliability of the ground truth.

In our earlier work [6], we compiled a list of typical cases that illustrate the pervasiveness of ground truth problems in software engineering and provided a prioritization and remediation process. We listed six different software engineering applications (CRR, reopened bug prediction, sentiment analysis, defect prediction, rework estimation, and BA) that potentially suffer from ground truth problems.

In this paper, we apply the ground truth improvement process and some of the improvement strategies described in Tuzun et al. [6] to both CRR and BA tasks. Specifically, we extend our previous work on the CRR application [39] to generalize our approach by adding BA as an additional application. In the CRR application, the labeling was done in real-time by parties directly involved with reviewer assignments. After recognizing that the original labelers could have been prone to cognitive biases, we looked for more objective, longer-term success factors in the data that confirm or refute the labeling decisions, and devised a heuristic to flag and remove the samples that violated the identified success factor. Cleaning up the ground truth data by removing suspect samples improved the performance of the evaluated CRR techniques. Here we describe the approach used in the CRR application, generalize it to task assignment with an additional application concerning the BA case.

3. Strategies for fixing the ground truth in task assignment

Our goal is to eliminate systematic labeling bias in historical data about task assignment. To be able to do this, we need a way to identify whether a task assignment decision can be considered as successful after the fact: that is, whether the labels of the assignment sample, the assignees, turned out to be the right choices for the task represented by that sample. Thus deciding label correctness requires a success measure. If the success measure is qualitative or subjective, we can only use manual methods to determine it. For example, for CRR, additional expert checks may be put in place to flag low-quality reviews when a PR is closed. The flagged assignments can then be excluded from the data. The same can also be done *post hoc*: the reviewer assignment data can be cleaned by an expert after the fact using similar quality checks. Similar manual corrections can be applied in BA. However, these expert-reliant approaches would not scale up well because of their cost, impracticality, or both. Besides, manual expert checks are subjective and can still be error-prone, and we must still assume the expert performing the checks and verifying the labels is impartial: if the expert is biased, we create a circular situation in which we attempt to address the bias by a method that may re-introduce the same or another kind of bias during the correction process.

In the following subsections, we focus on objective, automatically identifiable task assignment success measures for the two applications we are considering, and propose a heuristic to eliminate labeling bias based on these measures.

3.1. Characterization of a successful assignment task

In CRR case, when PRs are associated with coding tasks or defects in an issue or task tracking system, an objective success measure can be defined: if the tracked element is not reopened again, the PR is deemed *successful*. The PR process often involves multiple rounds of reviewers commenting on the scope of the PR, possibly resulting in several commits to address any outstanding concerns, and concludes with a merge into the main branch to close the PR out. The code review task associated with the PR itself is *successful* if the PR is successful. Similarly, the reviewers performing the code review for the PR themselves can be deemed *successful* if the PR they helped close out is successful.

A similar success measure can be defined for a bug fix. Identifying the appropriate developer to fix a bug is a crucial part of the software development lifecycle. The assigned developer is responsible for providing a fix for the reported problem. A *successful* bug assignee can be defined as a developer who provides a proper fix to the bug that has been assigned to them. We can first classify a bug fix as *successful* if the bug is not reopened after resolving the bug. If the bug fix is successful, we can conclude that the developer assigned to that bug was a *successful* bug assignee.

While the above success measures are objective, they still have a few caveats. First, it cannot be immediately determined whether a task assignment is successful: we can only decide the success status in retrospect, after sufficient time has elapsed following the closing of the PR or bug report. Second, they rely on the assumption that reopening the underlying report or task after the PR or report has been closed can only be due to the original action (or the person performing that action) not having been successful, and not due to, for example, something outside the scope of the original task having changed, causing a ripple effect. A typical situation that can cause such a ripple effect is a future commit that refactors a dependency associated with a development task. Third caveat is that the objective success measure assumes perfect bug identification: a new bug is created only when the origin of a newly discovered issue cannot reliably be traced to an existing bug report that can be reopened (no duplicate entries in the bug tracking system). We will accept these caveats, and see whether, even in their presence, the approach that we propose can achieve reasonable improvements in the performance of existing CRR and BA techniques with the adopted success measures.

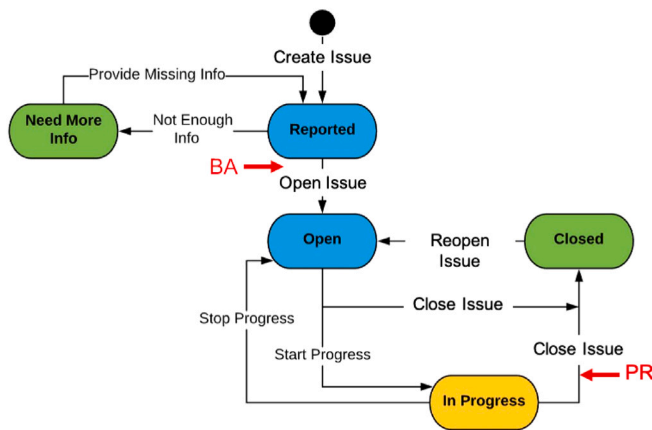


Fig. 1. The lifecycle of a bug and where it interacts with the BA and PR process.

3.2. The debiasing method

The objective success measures defined in the previous section provides us with a straightforward way to identify incorrect labels in CRR and BA datasets and remove them. For CRR, the success measure requires linkages between PRs and tasks logged in a tracking system. Most available datasets that have the required linkage focus on bug-related code review performed in the context of a PR. The CRR techniques on which we have tested our approach do the same. So in the CRR case, we limit ourselves to code reviews performed in the context of PRs associated with bugs. For BA, no such linkage is necessary since the success measure is directly associated with the status of a bug report.

Fig. 1 shows the typical flow of a bug as it is tracked in a bug tracking system (e.g., JIRA). The circles (*Reported, Need More Info, Open, In Progress, Closed*) represent the different states. BA happens in an earlier stage of a bug's lifecycle. After a bug is identified, the reporter should go through previous bug reports to see whether the bug encountered is a new bug or it already existed and was previously resolved in the bug tracking system. If no previous bug report matches the identified bug, a new bug report is created in the system. At this stage, the bug is in the *Reported* state. The BA process takes place in between the *Reported* and *Open* state, as the bug is considered *Open* when a developer is assigned to fix it. The PR process interjects this flow at the transition from the *In Progress* to the *Closed* state. After a developer fixes a bug, before the bug is closed, the developer creates a PR and one or more team members are invited to review the code. A PR conversation takes place discussing the fixes, which may result in a number of additional commits if the fixes were deemed inadequate. When the reviewers finally approve the fix, the developer merges the PR to the master and the bug's status is changed to *Closed.*

After merging, the patch is eventually deployed. However further testing, field testing by end users, or ongoing development work may at one point reveal that the bug had not been fixed as intended, in which case the bug may be reopened by changing its status back from *Closed* to *Open*. When this happens, barring other rare reasons that may cause a previously identified bug to reappear exactly in the same context, we can reclassify the original PR and bug report as having been unsuccessful: the assignee for a task assignment become a candidate for a biased, mislabeled sample, and thus for removal from the dataset.

The debiasing approach is based on the above reasoning, and consists of the removal of all potentially unsuccessful PRs for the CRR case and unsuccessful bug reports for the BA case. Given a CRR dataset consisting of a set of PRs, reviewer assignments for each PR, a bug associated with each PR, and the status history of each bug in the issue tracking system, we check, for each PR, whether the associated bug was reopened after it was closed following the merge. Similarly, given a BA dataset with historical bug reports, bug assignees for each report, and the status history of each bug, we check, for each closed bug report, whether the report was reopened at a later time. If in both cases, the given PR or bug report was reopened, we consider them to be unsuccessful *post hoc* and remove the associated samples from the data along with the task assignee. Next, we test through experiments whether this debiasing approach improves the performance of existing automated CRR and BA techniques that rely on historical task assignments.

4. Experiments and results

4.1. Dataset descriptions and preprocessing

4.1.1. The CRR dataset

For the CRR application, we evaluate the debiasing method on two different datasets. These datasets belong to projects from two sources: Qt,[1] a company that develops cross-platform software, and from Apache.[2] The projects are Qt Creator and HIVE, respectively. They are chosen because they are both open-source, have full PR and code review history, and the PR information is linked to the bug tracking information, as required.

For Qt Creator, we extracted the PR history[3] and bug history[4] until December 2019. For HIVE, we used the version provided by SEOSS 33 [49], a dataset repository that includes data retrieved from several open-source software projects. In the data gathering stage, we used the Perceval tool from GrimoireLab [50], which allows accessing datasets from both GitHub and Jira. Most of the PRs in the two datasets have Jira bug IDs (in HIVE 96.34%, and in Qt-Creator 73.18%). This allowed us tracing PRs to Jira bug reports.

Before we apply the debiasing method on these datasets, we performed three preprocessing steps. As a first step, we removed the PRs that do not have any association with a Jira bug. In both datasets, some seemingly distinct reviewer labels correspond to the same reviewer. For instance, two different reviewer labels "jkobus" and "Jarek Kobus" may refer to the same reviewer. In the second step of the preprocessing, duplicate reviewer labels were identified and merged automatically if they corresponded to the same email address. In the third step, we checked whether a PR's associated bug in the Jira database was reopened after the PR was merged. If so, we tagged these PRs as *unsuccessful.*

Table 1 shows the total number PRs, unsuccessful PRs, and the ratio of unsuccessful PRs to successful PRs. The total number of PRs in Qt Creator after preprocessing was 5927.406 of these PRs were unsuccessful, corresponding to a failure/success ratio of 7%. The HIVE dataset had 3621 PRs after preprocessing with 196 unsuccessful ones, yielding a failure/success ratio of 5%. There were 152 distinct code reviewers in the Qt Creator dataset and 108 distinct code reviewers in the HIVE dataset.

4.1.2. The BA dataset

For the BA application, we used four different datasets. Three of the datasets, Hadoop,[5] Netbeans[6] and Kafka,[7] are Apache[2] projects. The last dataset, Evergreen,[8] is a project from MongoDB,[9] a document-oriented database system. These projects are chosen because they are open-source, and provide full bug status history and bug assignee

[1] https://doc.qt.io/qt-5/index.html
[2] https://www.apache.org
[3] https://code.qt.io/cgit/playground/qt-creator/.
[4] https://bugreports.qt.io/projects/QTCREATORBUG/issues/.
[5] http://hadoop.apache.org.
[6] https://netbeans.apache.org.
[7] https://kafka.apache.org.
[8] https://github.com/evergreen-ci/evergreen.
[9] https://evergreen.mongodb.com/waterfall/evergreen.

Table 1
PR and reviewer statistics of the datasets.

Dataset	# Total PRs	# Unsuccessful PRs	U-to-S Ratio	# Reviewers
Qt Creator	5927	406	7%	152
HIVE	3621	196	5%	108

Table 2
BA statistics of the datasets.

Dataset	# Total Bugs	# Valid Bugs	# Unsuccessful BAs	U-to-S Ratio	# Assignees
Kafka	11 918	6457	270	4.2%	573
Hadoop	15 575	5635	289	5.1%	476
Netbeans	5705	1252	32	2.6%	141
Evergreen	10 545	6438	229	3.6%	62

information. The bug history of Hadoop, Netbeans, and Kafka were extracted from Jira until August 2021 using the Perceval [50] tool. For Evergreen, we used the version provided by Qamar et al. [51].

Before applying the debiasing method on these datasets, we performed certain preprocessing steps. First, we checked whether each assignee label was unique. We did not find any bug assignee labels that corresponded to the same developer. The second step was separating out the bugs that did not have an assignee. We expected any bug that had been in the Open state at one point in their lifecycle to have had an assignee, but we still detected bugs that had been opened at one point and did not have an assignee. After detecting these bugs, we separated them from the bugs that had a valid assignee. In the third step, we removed all bugs that were not closed at the time of data retrieval: the BA techniques we used to evaluate our debiasing approach train their classifiers on closed bugs. In our final preprocessing step, we identified all bugs that were reopened at one point, and tagged them as *unsuccessful*.

Table 2 shows the total number of bugs, the number of valid bugs, the number of unsuccessful BAs, the ratio of unsuccessful BAs to successful BAs, and the number of assignees in each dataset. Valid bugs are closed bugs with valid assignee labels. The number of unsuccessful BAs include only valid bug reports. For example, out of the 15,575 bugs in the Hadoop dataset, 5635 were closed and had a valid assignee label. Out the remaining 9940 bugs, 6074 were opened with assignee labels, 3235 were opened without assignee labels, and 631 were closed without assignee labels. Out of the 5635 valid bug reports, 289 were tagged as unsuccessful BAs, corresponding to a failure/success ratio of 5.1%. There were 476 distinct bug assignees in the dataset.

4.2. Evaluation setup

4.2.1. CRR evaluation

To evaluate the reliability and usefulness of the debiasing method, we selected five different CRR techniques from the literature, namely, Profile-based [27], RSTrace [15], Naïve Bayes, k-NN (5-NN), and Decision Tree. Initially, we wanted to apply the method to all CRR techniques discussed under Section 2. However, only a few of the CRR techniques [15,27] provide source code or pseudocode. Therefore, we selected those that we could actually run or re-implement. We had to implement the profile-based technique ourselves since the source code was not available. For RSTrace, we used the available implementation shared in the original paper [15]. For Naïve-Bayes, 5-NN, and Decision Tree, we used the implementations provided by the Scikit-learn library [52].

For the three machine learning techniques (Naïve Bayes, k-NN, and Decision Tree), we used the file-paths given in the PRs as features. To convert these file-paths to numeric values for classification, we applied two vectorizers (CountVectorizer and TfidfVectorizer) from the Scikit-learn library. Hyperparameters of the machine-learning-based models were optimized using values from commonly accepted ranges and according to the characteristics of our datasets. These values are given in Table 3.

Table 3
List of hyperparameter values for the learning-based CRR models.

Model	Hyperparameter	Values
Naïve-Bayes	Distribution type	{multinomial, Gaussian, Bernoulli}
5-NN	Distance type	{Manhattan, Euclidean}
	NN search algo.	{ball tree, KD tree, brute-force search}
	Weight function	{uniform, inverse distance}
Decision Tree	Split strategy	{Gini impurity, entropy}
	Measure of split quality	{best, random}
	Maximum depth	$\{1, 2, \ldots, 10\}$

Table 4
List of the hyperparameters for the learning-based BA models.

Model	Hyperparameter	Values
CNN Triage	Learning rate	{0.005, 0.01}
	Number of filters	{32, 64}
	Test size	{0.10, 0.20, 0.25}
	Evaluation Step	{13, 32}
Deep Triage	Batch size	{16, 32, 64, 128}
	Learning rate	{1e−3, 1e−4, 1e−5}
	Epoch	{5, 10, 100}

4.2.2. BA evaluation

We selected two different BA techniques, Deep Triage [36] and CNN Triage [29], from the literature to evaluate the debiasing method. The selected techniques are both deep-learning-based approaches. We wanted to evaluate our debiasing method on a variety of different techniques listed under Section 2. However, the source code for most of these techniques were not made available to the public.

Deep Triage [36] takes bug title and description as input and maps this information to individual bug assignees by using an attention-based, DBRNN-A with Word2Vec word embedding [53]. The classifier is trained with a dataset consisting of closed bugs with valid assignee labels. The technique also needs to learn bug representations from the bug summary and description of all samples. The model architecture involves nine layers. Input layer is followed by Embedding, Long-Short Term Memory (LSTM), Soft Attention, Batch Normalization, Merge, Dense, Dropout and finally another Dense layer.

CNN Triage [29] experiments with different word embedding techniques(Word2Vec, GloVe, ELMo) combined with a Convolutional Neural Network (CNN) to perform a bug assignment. In our experiments, we have used the Word2Vec model as the word embedding technique [53]. The classifier is trained with closed bugs with valid assignee labels. The bug description is fed to the neural network to recommend the appropriate bug assignee. After the text input is vectorized, it is passed through Convolutional and Pooling layers where the number of filters is taken as a hyperparameter. Finally, the data is passed through a Softmax Regression layer which gives the assignment probability of all developers for the input bug.

Before evaluating the debiasing method with these techniques, we ensured that the techniques were working correctly by training the models with their original datasets and replicating the original authors' results. We then tried the techniques on our datasets with different hyperparameter values to choose the best combination for each dataset. To perform the comparison, we subsequently retrained the models with the same chosen hyperparameter values using the debiased versions of each dataset to ensure the comparison is fair. Table 4 shows the hyperparameters and their tried values for each technique and dataset.

4.3. Performance measures

We assessed the accuracy of the CRR and BA techniques selected by two widely used measures: Top-k accuracy (namely, Top-1, Top-3 and Top-5) [54] and Mean Reciprocal Rank (MRR) [55]. A task assignment technique returns an ordered list of assignees for an input task, however only one of the returned assignees is the right one. Given a set of input tasks, Top-k accuracy is the probability that the right assignee occurs among the top k assignees returned for those input tasks. Reciprocal Rank is the inverse of the position of the right assignee for an input task. MRR is the average of these inverse positions over all input tasks.

We computed the relative improvements in the above measures after debiasing to demonstrate the effectiveness of our approach as follows:

$$\frac{S_{\text{after}} - S_{\text{before}}}{S_{\text{before}}}, \quad (1)$$

where, S_{before} is the performance of a task assignment technique before we applied debiasing and S_{after} is the performance after we applied debiasing using the same dataset.

4.4. Balancing

We balanced the samples in the CRR datasets with respect to successful and unsuccessful task assignments using under-sampling. However we did not do this with the samples in the BA datasets. The BA techniques used to evaluate the debiasing method are based on deep neural network (DNN) architectures. DNNs are more sensitive to false labels than other machine learning techniques, with a tendency to easily memorize noisy samples [56]: they suffer from excessive over-fitting when the models are trained with false labels. The over-fitting problem, when present, is often resistant to regularization techniques [56]. When measures are taken to remove false labels before training, as we do here with the debiasing method, their performance can be significantly improved even when the ratio of the false labels removed in the original dataset is low. In addition, DNN-based techniques are less tolerant to under-sampling than non-DNN-based techniques because they require a large number of samples to work. Therefore we decided not to apply balancing (through under-sampling) to the BA datasets, which are trained with DNN-based techniques.

The selected CRR techniques were trained and tested with the HIVE and Qt Creator datasets. Both datasets had a quite low unsuccessful PR rate (5% to 7%) according to our success criterion. The literature [57] suggests unsuccessful PRs are significantly more pervasive than they were in our datasets. The actual ratios of re-instated tasks in some popular open-source projects are much higher than the ratios measured in the CRR datasets. For example, in Eclipse, the ratio was found to be 16.1% and in OpenOffice as high as 26.31%. A lower than actual ratio often stems from the common practice of creating a new task assignment out of convenience because searching for existing reports, identifying the one that matches the new task, and reopening the existing task may require effort. Developers may not remember or know about the original task due to turnover or time lapse, and inadvertently mistake a recurring task for a brand new one.

The problem with a low unsuccessful PR rate in the dataset due to missed reopened tasks—bug reports in our case—is that debiasing through the removal of the corresponding data points would yield only marginal improvements in performance with non-DNN-based techniques. We are interested in assessing how much improvement can be achieved by debiasing if the training dataset's ratio was realistic, closer to the actual ratios observed in real practice. Therefore, we under-sampled the data [58] by randomly removing successful PRs until the unsuccessful PR ratio was in the same ballpark range as the rates reported in the literature: starting from the first successful PR, we randomly removed three out of every four successful PRs. This effectively quadrupled the unsuccessful PR ratios, bringing them closer to the more commonly observed values.

4.5. The evaluation process

Fig. 2 illustrates the evaluation process. The box Original (Og) represents the preprocessed dataset containing successful (S) and unsuccessful (U) PRs. Debiasing removes unsuccessful PRs, resulting in a Debiased (Db) dataset. At the first step, a task assignment technique is trained with both Og and Db datasets, resulting in two models. The performance of the models is compared to assess the effect of debiasing. We expect the performance of the model trained with the debiased dataset to be better since debiasing attempts to remove samples suspected of having bad labels.

When the original datasets contain too few samples that exhibit possible labeling bias and the techniques used are tolerant of false labels, we expect the improvement to be marginal, as explained in the previous section. In these situations, to see how much improvement can be achieved when systematic labeling bias is as pervasive as it is reported in the literature, we balanced the PRs to bring the ratio of the unsuccessful PRs to realistic levels. This is the Balanced (Ba) dataset at the top left. The Ba dataset is then debiased by the same procedure as before, resulting in the Balanced-Debiased (DbBa) dataset. The results are again compared for the evaluated task assignment technique. Our expectation is the improvement in performance in this latter comparison to be superior to the one without balancing.

Finally, an extra validation step is possible to check any observed relative improvement in performance is not due to a random reduction in the sample size, but due to the targeted removal of only badly labeled (unsuccessful) PRs: for this to be true, random removal of datapoints instead of targeted removal should not improve the performance. We form several datasets by randomly removing data from the successful subset only (favoring this class for the reduction) and from both successful and unsuccessful PRs (not favoring any class). These reductions give rise to the Reduced-Biased (RBi) and Reduced-Unbiased (RUb) datasets shown at the bottom corners of Fig. 2. We compare the performance of the task assignment technique with these datasets to the performance with the dataset DbBa to show that any improvement in performance with debiasing is not merely accidental, but must be because of having deleted the badly labeled samples. Like balancing, we conducted this validation step only for the CRR cases: therefore, the BA application only relied on the Original(Og) and Debiased(Db) datasets in Fig. 2.

4.6. Testing strategies

4.6.1. CRR testing strategies

While testing both categories of techniques, we preserved the chronological order of the data to avoid attempting to predict past instances using future instances. Therefore, we made sure that the training samples always preceded the tested samples.

While evaluating the debiasing method on the optimization-based CRR techniques (Profile-based and RSTrace), we incrementally predicted each sample using all samples that preceded it, expanding the models used for prediction one sample at a time.

However this fine-grained, one-by-one strategy cannot be applied in learning-based approaches: the standard testing methodology is based on using multiple folds. Therefore, for the learning-based techniques (Naïve Bayes, k-NN [5-NN], and Decision Tree), we performed sliding-window testing. We divided our dataset into 10 folds, chose a test fold, and trained our models using all the folds chronologically located before the test fold. In each iteration, we shifted the window to the next fold and repeated the same procedure.

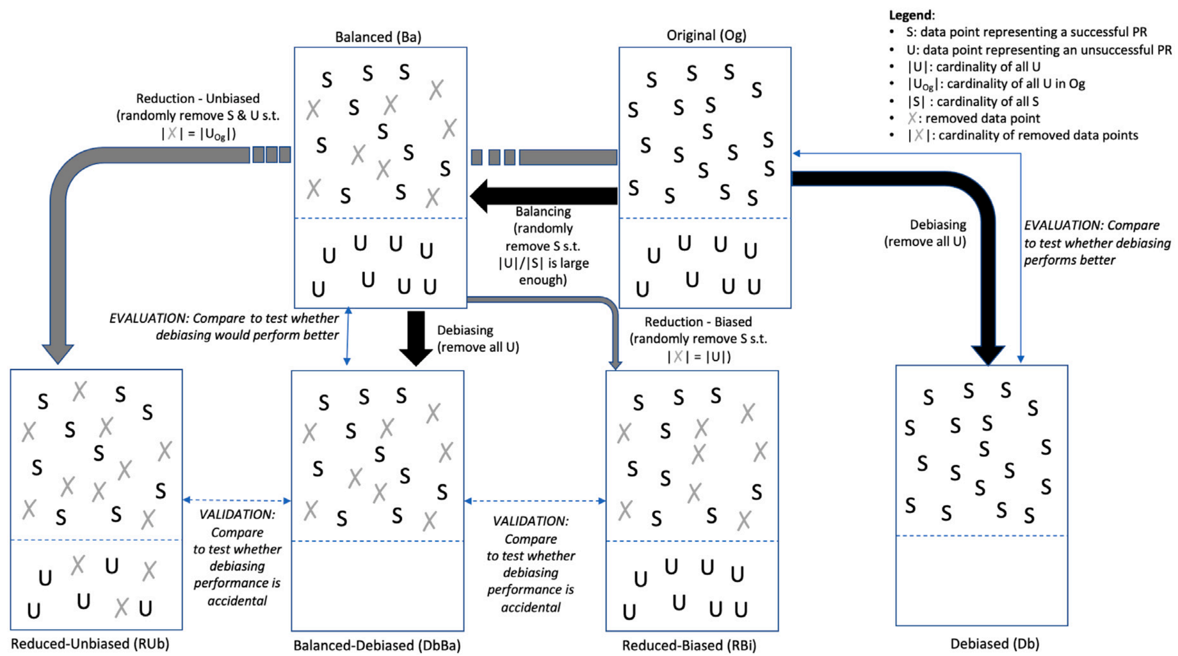


Fig. 2. The evaluation process using different versions of the preprocessed datasets.

4.6.2. BA testing strategies

While testing the BA techniques, we ensured that the chronological order of the data was preserved. The Deep Triage already preserved the chronological order of the data. However, in CNN Triage, the data was being shuffled. Therefore, we modified the source code to disable the shuffling of data to preserve the chronological order. Both Deep Triage and CNN Triage originally used only Top-k accuracy as their performance measure. We modified the implementation to also measure MRR.

4.7. Results: Effect of debiasing

4.7.1. CRR results

In order to investigate the effect of removing unsuccessful PRs, we evaluated the performance of the five CRR techniques on Qt Creator and HIVE datasets before and after the debiasing. Notice that after debiasing, the datasets contained only samples corresponding to successful PRs.

Table 5 summarizes the Top-3 and Top-5 accuracies before and after the debiasing using both datasets. The three learning-based CRR techniques perform better with debiasing than the two optimization-based ones according to the Top-3 measure. For the Naïve Bayes technique, we observe the highest relative improvement, 17% and 23%, on the HIVE and Qt Creator datasets, respectively.

The Top-1 and MRR results are compared in Figs. 3 and 4, respectively. Overall, higher accuracy rates were obtained for higher k values for Top-k accuracy. We did not observe a considerable improvement in the accuracy of optimization-based approaches for any k value through debiasing, whereas learning-based approaches showed a clear improvement especially in Top-3 accuracy.

Unlike the Top-k measures that focus on best predictions, the MRR measure considers the ranking of each prediction. It is therefore more representative of average performance. Fig. 4 shows the MRR scores before and after the debiasing. In terms of MRR, debiasing yields the best relative improvement on the learning-based techniques. The improvement for the 5-NN technique on the HIVE dataset is 25% and for Naïve Bayes technique on Qt Creator dataset is 26%. These improvements are higher than those observed with the Top-k measures.

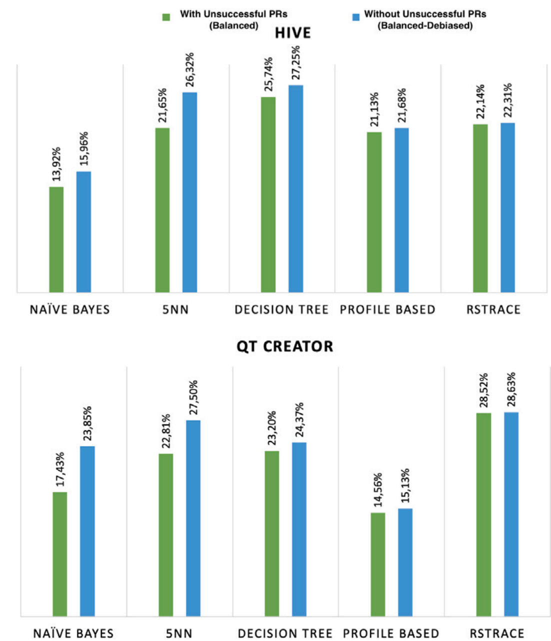


Fig. 3. CRR Top-1 Accuracy before and after debiasing on the balanced versions of the datasets.

4.7.2. BA results

To study the effect of removing unsuccessful bugs from the datasets, we evaluated the accuracy of the two BA techniques on Kafka, Hadoop, Netbeans, and Evergreen datasets before and after the debiasing. Table 6 summarizes the Top-3 and Top-5 accuracies before and after the debiasing for all datasets. Both BA techniques performed better according to the Top-3 and Top-5 measures on the debiased datasets. The highest relative improvement is observed in the Top-3 accuracy of CNN Triage on the Hadoop dataset. The Top-3 accuracy of the CNN Triage technique on the Hadoop dataset showed 85.61% relative improvement after debiasing. The highest improvement in the Top-5 accuracy was achieved in the CNN Triage technique on the Kafka

Table 5
Performance of CRR techniques before and after debiasing on the balanced versions of HIVE and Qt Creator.

Technique	Top-3 accuracy					
	HIVE			Qt Creator		
	Before debiasing	After debiasing	Relative improvement	Before debiasing	After debiasing	Relative improvement
Naïve Bayes	23.52%	27.72%	17.86%	26.23%	32.60%	24.29%
5-NN	34.20%	38.99%	14.01%	42.54%	48.28%	13.49%
Decision Tree	37.51%	40.74%	8.61%	40.94%	44.75%	9.31%
Profile based	39.18%	39.44%	0.66%	39.74%	40.03%	0.73%
RSTrace	40.78%	41.12%	0.83%	42.36%	42.67%	0.73%
Technique	**Top-5 accuracy**					
	HIVE			Qt Creator		
	Before debiasing	After debiasing	Relative improvement	Before debiasing	After debiasing	Relative improvement
Naïve Bayes	27.75%	29.31%	5.62%	35.59%	39.53%	11.07%
5-NN	47.12%	50.20%	6.54%	52.71%	55.77%	5.81%
Decision Tree	46.69%	50.71%	8.61%	53.77%	55.37%	2.98%
Profile based	50.30%	51.13%	1.65%	48.80%	49.44%	1.31%
RSTrace	48.20%	48.34%	0.29%	50.40%	50.57%	0.34%

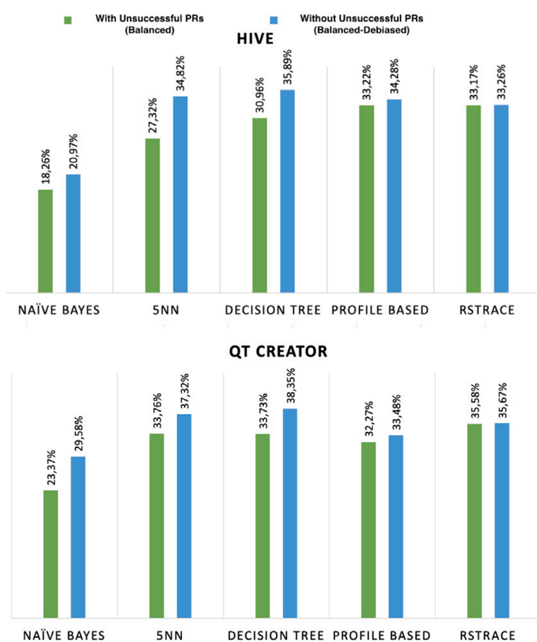


Fig. 4. CRR MRR Scores before and after debiasing on the balanced versions of the datasets.

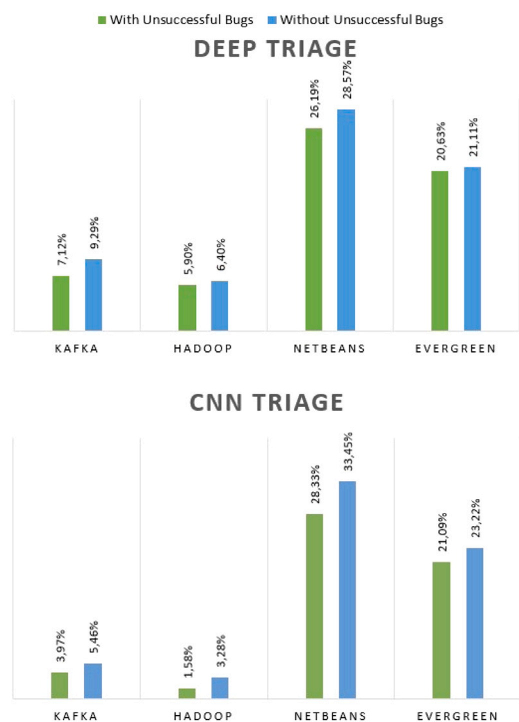


Fig. 5. BA Top-1 Accuracy before and after debiasing on the datasets.

dataset with 35.13% relative improvement after debiasing. Fig. 5 summarizes the Top-1 accuracies of both techniques on all datasets. While the increase in Top-1 accuracies is less than those of Top-3 and Top-5, the improvement is still non-trivial. Both BA techniques achieved lower accuracies overall for the Hadoop and Kafka datasets due to the larger number of bug assignees in these datasets.

The results for MRR before and after debiasing is shown in Fig. 6. Overall, the MRR scores of the techniques increased after debiasing. The greatest MRR improvement, at 67%, occurred with CNN Triage on the Evergreen dataset.

5. Discussion

The debiasing method we propose can be applied to any assignment task in software engineering (i.e., feature assignment, test assignment tasks). However, there are three prerequisites that need to be satisfied to successfully apply the method for a software engineering assignment task. (1) Previous task assignment data should be readily available to represent the ground truth. (2) There should be an objective success measure (e.g., in the bug assignment problem, the success measure was about whether a bug was reopened) to assess if the assignment instance is successful. (3) Unsuccessful and successful assignment samples should be balanced for most machine learning techniques to perform sufficiently well.

In Section 5.1 we revise two research questions and explain our approach to them. Afterwards, in Section 5.2 we discuss that the proposed debiasing method is really effective. Later, in Section 5.3 we show the number of unsuccessful samples could be superfluous and confirm reopened PRs indeed had poor reviews. Lastly, in Section 5.4 we consider internal and external threats to validity.

5.1. Research questions

RQ1: *How can we eliminate systematic labeling bias in task assignment ground truth data?*

Since manual methods are not cost-efficient, are error-prone, and do not scale up well, we looked for an automated method based on

Table 6
Performance of BA techniques before and after debiasing on Kafka, Hadoop, Netbeans and Evergreen.

Dataset	Top-3 accuracy					
	DeepTriage			CNNTriage		
	Before debiasing	After debiasing	Relative improvement	Before debiasing	After debiasing	Relative improvement
Kafka	17.48%	19.81%	13.33%	9.72%	12.85%	32.20%
Hadoop	12.92%	13.77%	6.58%	6.67%	12.38%	85.61%
Netbeans	38.89%	40.61%	4.42%	44.33%	48.46%	9.32%
Evergreen	31.78%	33.26%	4.66%	40.08%	42.96%	7.19%
Dataset	Top-5 accuracy					
	DeepTriage			CNNTriage		
	Before debiasing	After debiasing	Relative improvement	Before debiasing	After debiasing	Relative improvement
Kafka	25.50%	27.58%	8.16%	13.95%	18.85%	35.13%
Hadoop	18.11%	19.09%	5.41%	14.20%	17.84%	25.63%
Netbeans	44.84%	48.57%	8.32%	51.33%	56.31%	9.70%
Evergreen	41.10%	44.39%	8.00%	52.37%	54.43%	3.93%

Table 7
MRR for random vs. Targeted removal of PR samples.

	HIVE				
	Naive Bayes	5-NN	Decision Tree	Profile based	RSTrace
With unsuccessful PRs (Balanced)	18.26%	27.32%	30.96%	33.22%	33.17%
Without unsuccessful PRs (Balanced-Debiased)	20.97%	34.82%	35.89%	34.28%	33.26%
Without unsuccessful–successful PRs (Reduced-Unbiased)	18.13%	26.62%	30.75%	33.14%	33.09%
Without successful PRs (Reduced-Biased)	17.27%	23.24%	26.83%	32.56%	32.87%
	Qt Creator				
	Naive Bayes	5-NN	Decision Tree	Profile based	RSTrace
With unsuccessful PRs (Balanced)	23.37%	33.76%	33.73%	32.27%	35.58%
Without unsuccessful PRs (Balanced-Debiased)	29.58%	37.32%	38.35%	33.48%	35.67%
Without unsuccessful–successful PRs (Reduced-Unbiased)	22.12%	33.49%	32.23%	32.15%	35.41%
Without successful PRs (Reduced-Biased)	18.30%	30.72%	29.41%	31.43%	35.13%

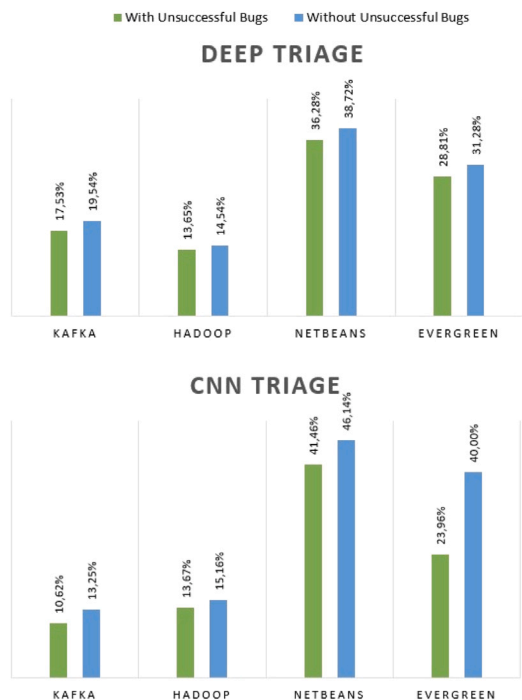


Fig. 6. BA MRR Scores before and after debiasing on the datasets.

an objective success measure to remove suspicious labels. In CRR, a PR, the underlying code review, and the assigned reviewer's work were deemed *successful* only if the bug the PR targeted was never reopened following a successful merge and the associated closure of the bug. Similarly in BA, the bug fix and the assigned developer's work were deemed *successful* only if the associated bug report was never reopened following its closure. The debiasing method we proposed simply removed these deemed *unsuccessful* samples from the task assignment data to eliminate possible biases in past task assignments. These success measures can easily be adopted to other task assignment problems in software engineering, for example for assigning new features to appropriate developers. Task assignment techniques could then use the debiased data as their ground truth to build their models for improved performance.

RQ2: *How does systematic labeling bias elimination in the ground truth data affect the performance of task assignment techniques that rely on the data?*

We applied our automated debiasing method to a diverse set of five CRR techniques using two open-source datasets and two BA techniques using five open-source datasets. We found that, when the data had sufficiently high-rates of badly labeled samples, the performance of both CRR and BA techniques in general improved after debiasing. The highest improvement in CRR was observed with learning-based CRR techniques. The improvements in the optimization-based CRR techniques tested were marginal. Both BA techniques used to evaluate our debiasing approach were learning-based, and exhibited improvement levels that exceeded the ones observed for the learning-based CRR techniques.

We can speculate about the reasons behind the improvement difference between optimization-based and learning-based CRR techniques. The reviewer recommendation heuristics of the optimization-based techniques tested did not as heavily depend on learning from the past data as those of learning-based techniques. Additionally, not every sample may have been as equally valuable in the optimization-based techniques and the optimization criterion may have inadvertently already discounted badly labeled samples. Therefore, a debiasing approach focused on removing suspected badly labeled samples may not have made much difference in these techniques.

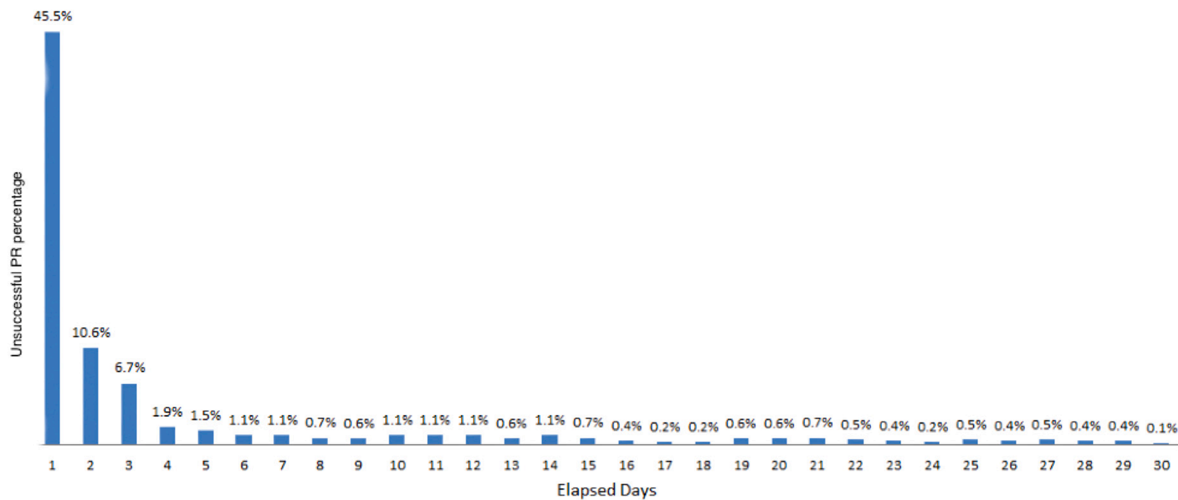


Fig. 7. Percentage of unsuccessful PRs detected at each day following a successful merge due to reopening of the underlying bug (HIVE dataset).

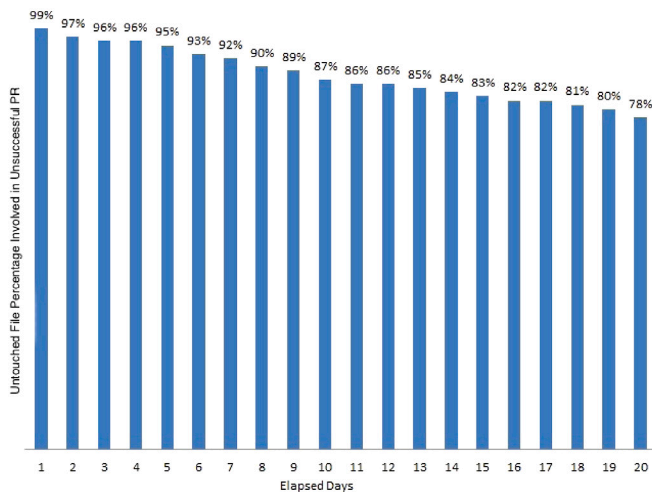


Fig. 8. Percentage of untouched PR files after closure over time (HIVE dataset).

The higher improvements with DNN-based architectures used in BA can be explained by the higher sensitivity of the DNN architectures to false labels compared to the sensitivity of other machine learning techniques used in CRR. Debiasing would naturally yield better result with techniques that more easily degrade with false labels.

As mentioned in Section 3.2, ideally, when a bug is found, the reporter should go through the past bug reports to identify whether the observed bug has been encountered before. If the bug had already been identified in the past and was closed, the bug report needs to be reopened. However, this process may not be followed in open-source projects due to laxer rules, typically high number of developers contributing to the project, and the time consuming nature of this practice. Instead, the reporters may be more inclined to create a new bug report for an already encountered bug. With industrial projects, we would expect a more structured quality assurance process and more compliance with stricter guidelines, which should produce a higher quality data with less noise and more reliability. Therefore, we would expect the reopen rate in industrial projects to be higher and our debiasing method to be more effective.

Based on testing our debiasing method with two task assignment applications, multiple task assignment techniques for each application, and multiple datasets, we conclude that the method improves the quality of the ground truth, and is especially worthwhile for learning-based task assignment techniques.

5.2. *Does debiasing actually work or is it just coincidence?*

To illustrate that improvements observed with debiasing are not accidental, we performed an additional validation step with the CRR datasets and techniques. We used the balanced versions of the datasets for this purpose because the original, unbalanced versions showed only marginal improvements with debiasing due to the very small number of badly labeled samples (as explained in Section 4.4). We used the following procedure for this extra validation step:

- Remove PRs randomly from a dataset and evaluate the accuracy of each technique before and after debiasing. This process compares the dataset version RUb (Reduced-Unbiased) in Fig. 2 to the version DbBa (Balanced-Debiased). The number of samples removed equals the number of unsuccessful PRs to achieve a fair comparison. This is repeated 100 times to create different randomly reduced versions of a dataset to compare with the DbBa version. The performance results from the different random reductions are then averaged.
- Randomly remove only successful PRs from a dataset (thus introducing a bias against successful PRs) and evaluate the accuracy of each technique before and after debiasing. This process compares the dataset version RBi (Reduced-Biased) in Fig. 2 to the version DbBa (Debiased-Balanced). Again the random reduction is repeated 100 times and performance results averaged.

Table 7 summarizes the results of this step for the MRR measure. The results are expected. All techniques performed best without unsuccessful PRs (DbBa) and worst with randomly removed successful PRs (RBi). In all cases, the techniques invariably performed worse with randomly removed PRs (RUb) than without unsuccessful PRs (DbBa). We conclude that the improvements observed with debiasing is not accidental since targeted removal of samples focusing on unsuccessful PRs always gave better results.

5.3. *How many unsuccessful samples could be superfluous?*

In Section 3.1, we discussed a number of caveats about the success measure adopted. We now focus on the second caveat, and assess, in the context of the CRR application, to what extent reopened review tasks could be attributed to reasons other than the review/reviewer quality of the original associated PR. This assessment considers the possibility that the original PR was indeed successful and the reopened task was in fact a false positive. The false unsuccessful attribution could be due to changes to the codebase unrelated to the original task

(e.g., changes in the underlying dependencies that makes it look like the original bug suddenly resurfaced instead of being reported as a new bug). To do this, we analyzed the elapsed time between *Closed* and *Reopened* transitions of each reopened bug in the datasets. If the elapsed time was small, e.g., less than one day, resurfacing of the bug would be unlikely to have been related to external circumstances impossible to have been detected by the PR reviewers. We also looked at the percentage of the files involved in an unsuccessful PR that were not changed (untouched) in a commit following the closure of the associated bug. If this percentage was high after a certain time had elapsed, their likelihood of causing the bug to resurface was considered low, hence the PR was likely genuinely unsuccessful.

Fig. 7 shows the above effects in the HIVE dataset: 45% of the bugs were reopened on the same day of the PR and 80% of the bugs were reopened within 24 days. Fig. 8 shows that 80% of the files involved in an unsuccessful PR remained untouched 19 days after the closure of the underlying bug. The results were similar on the second dataset. Because most bugs were reopened in the first few days of their closure and most files involved in an unsuccessful PR remained untouched during the initial days after closure, we believe mislabeling unsuccessful PRs, although possible in rare circumstances, was unlikely to have been pervasive enough to compromise the PR success measure.

To confirm that reopened PRs indeed had poor reviews, we performed a quality analysis of in Qt Creator dataset for a random 10% sample (40 data points). You can find this analysis in a supplement posted to Figshare.[10] Two authors independently inspected the quality of each review for this sample. We found that 32 out of the 40 (80%) had in fact poor reviews according to the criteria we used. We categorized poor reviews as *Superficial* (only "Looks-Good-to-Me" style comments, or missing comments) (21), *Overruled* (author indicated reviewer had misjudged the changes) (3), and *Poor-Effort* (self-admittance of a substandard/rushed review) (6).

5.4. Threats to validity

Our method of identifying a badly labeled task assignment is subject to a construct threat [59]. It may not be possible to catch all reopened bugs: some may have been completely missed, others may be reopened in the future, and thus may not have been captured in the dataset. Leaving these false negatives in the dataset would reduce the efficacy of debiasing. Conversely, there may have been false positives: a task assignment identified as unsuccessful due to a reopened bug may have actually been successful. We discussed some possible reasons for such cases in Section 5.3, and by examining two indicators in the datasets for one of the applications, concluded that these cases are likely to have been rare.

CRR datasets had low unsuccessful PR rates: possibly, in many cases, the developers who created the bug reports failed to associate a recurring bug with an existing bug report, and instead created a new bug report (we previously discussed possible reasons for this.) At these low rates, the introduced bias is not significant, and removing it would not yield much benefit. We observed precisely this effect in the original data: improvements in performance measures were less than 1%, and thus not material. We thus looked at the reopened bug rates in the literature and balanced the CRR datasets by randomly removing successful PRs to move the unsuccessful PR ratio to within the reported ranges, and evaluated the debiasing method with these reduced datasets. Because of this adjustment, we must accept that any observed improvements are conditional on a dataset having sufficient systematic labeling bias.

BA techniques we used in this study were not used out of the box. We had to modify the published source code for each technique to be able to run them with shell commands. Although our modifications should not have affected how the techniques work, there remains a small risk that we may have inadvertently changed their original behavior. Although we were conducting experiments on a server with 256 GB memory and 64-core processor, while running the CNN Triage technique, we encountered memory limitation problems caused by word embeddings: this narrowed down the range of values we could use for the hyperparameter tuning.

Threats to external validity are concerned with generalizability [59]. We evaluated our approach on two open-source datasets and five different CRR techniques and four datasets with two BA techniques. We believe the CRR and BA techniques used are a reasonable representation of common approaches. However we acknowledge the limitations of two datasets for CRR application and two techniques for the BA application. We could not find additional CRR datasets that both contained sufficient samples and integrated PR information with bug tracking, as required. We could not find any other BA techniques with published source code or an executable.

Open-source projects typically have high turnover rates. Unlike in closed-source projects, many contributors and reviewers become inactive over time, and new contributors constantly join these projects. For this reason, the number and activity level of the assignees and the nature and difficulty of development task assignments may be drastically different in closed-source projects. Therefore, our findings may not apply to closed-source projects.

One way to mitigate a study's internal threats is to make the study replicable. We provide a complete replication package that includes the datasets, source code modifications, all of our scripts, and step-by-step evaluation procedures in a Figshare archive[10].

6. Conclusion and future work

Good assignee selection is central to effective task assignment in software development. Task assignment techniques attempt to automate the assignee selection process, but many techniques build their models and evaluate them using historical data whose ground truth may be unreliable. Ground truth problems often result from the susceptibility of human decision makers to cognitive biases, such as substituting a convenience attribute for a competence attribute. When the task assignments are associated with a success measure, such as the closing of a bug report indefinitely, we can use the success measure to detect, *post-hoc*, unsuccessful assignments, and remove the associated samples from the historical data. We used this method to clean up, or *debias*, the ground truth data in two task assignment applications: CRR and BA.

To evaluate our debiasing method, we conducted experiments with five existing CRR techniques (Profile based, RSTrace, Naïve Bayes, k-NN [5-NN], Decision Tree) on two open-source datasets (HIVE, Qt Creator) and two existing BA techniques (CNN Triage, Deep Triage) on four open-source datasets (Kafka, Hadoop, Netbeans, Evergreen). We compared the MRR and Top-k performance of these techniques both with and without debiased samples, and found that debiasing using our method can achieve performance improvements between 10%–26% with general learning-based techniques, and 7%–67% for deep-learning-based techniques. However, the debiasing method was not able to achieve significant levels of improvement with optimization-based techniques. Unfortunately, it is not possible to set a universal threshold for how much improvement would be considered significant or worthwhile in an arbitrary task assignment context. Since our debiasing approach is automated with a negligible marginal cost, even small levels of improvement would be welcome.

Our work has implications for both practitioners and researchers. Researchers can apply our proposed debiasing approach preprocessing the training data to improve the accuracy of their learning-based task assignment models. Recommendation tools built on these models would then inherit these improvements. The debiasing method could also be

[10] https://figshare.com/s/f456ae85508bb1bfa78e.

useful for flagging potentially improper development task assignments in an organization.

In the future, we are planning to extend our evaluation to other task assignment problems, for example, the assignment of new software requirements to an appropriate member of a development team, using additional datasets. If expanded evaluations prove the debiasing method to be widely effective, we plan to provide tool support for automated debiasing so that our method can easily be integrated into existing recommendation systems. We also plan to evaluate our approach on industrial projects since such projects could provide a better use case with less noise for our approach.

CRediT authorship contribution statement

K. Ayberk Tecimer: Methodology, Software, Investigation, Data curation, Writing – original draft, Visualization. **Eray Tüzün:** Conceptualization, Validation, Resources, Writing – review & editing, Supervision, Project administration. **Cansu Moran:** Methodology, Software, Investigation, Data curation, Writing – original draft, Visualization. **Hakan Erdogmus:** Conceptualization, Validation, Writing – review & editing, Supervision.

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] R. Smith, J. Hale, A. Parrish, An empirical study using task assignment patterns to improve the accuracy of software effort estimation, IEEE Trans. Softw. Eng. 27 (3) (2001) 264–271, http://dx.doi.org/10.1109/32.910861.

[2] C. Hannebauer, M. Patalas, S. Stünkel, V. Gruhn, Automatically recommending code reviewers based on their expertise: An empirical comparison, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, in: ASE 2016, Association for Computing Machinery, New York, NY, USA, 2016, pp. 99–110, http://dx.doi.org/10.1145/2970276.2970306.

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

[4] B. Turhan, A. Tosun Mısırlı, A. Bener, Empirical evaluation of the effects of mixed project data on learning defect predictors, Inf. Softw. Technol. 55 (6) (2013) 1101–1118, http://dx.doi.org/10.1016/j.infsof.2012.10.003.

[5] C. Weiss, R. Premraj, T. Zimmermann, A. Zeller, How long will it take to fix this bug? in: Proceedings of the Fourth International Workshop on Mining Software Repositories, in: MSR '07, IEEE Computer Society, USA, 2007, p. 1, http://dx.doi.org/10.1109/MSR.2007.13.

[6] E. Tuzun, H. Erdogmus, M.T. Baldassarre, M. Felderer, R. Feldt, B. Turhan, Ground truth deficiencies in software engineering: when codifying the past can be counterproductive, IEEE Softw. (2021) http://dx.doi.org/10.1109/MS.2021.3098670.

[7] C. Sadowski, E. Söderberg, L. Church, M. Sipko, A. Bacchelli, Modern code review : A case study at google, in: Proceedings of the 40th International Conference on Software Engineering Software Engineering in Practice - ICSE-SEIP 18, 2018, http://dx.doi.org/10.1145/3183519.3183525.

[8] Google, Code review developer guide, 2020, URL: https://github.com/google/eng-practices/blob/master/review/index.md.

[9] V. Balachandran, Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 931–940, http://dx.doi.org/10.1109/ICSE.2013.6606642.

[10] J.B. Lee, A. Ihara, A. Monden, K. Matsumoto, Patch reviewer recommendation in OSS projects, in: 2013 20th Asia-Pacific Software Engineering Conference (APSEC), Vol. 2, IEEE, 2013, pp. 1–6, http://dx.doi.org/10.1109/APSEC.2013.103.

[11] P. Thongtanunam, C. Tantithamthavorn, R.G. Kula, N. Yoshida, H. Iida, K.-i. Matsumoto, Who should review my code ? in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, 2015, pp. 141–150, http://dx.doi.org/10.1109/SANER.2015.7081824.

[12] X. Xia, D. Lo, X. Wang, X. Yang, Who should review this change ? in: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2015, pp. 261–270, http://dx.doi.org/10.1109/ICSM.2015.7332472.

[13] A. Ouni, R.G. Kula, K. Inoue, Search-based peer reviewers recommendation in modern code review, in: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2016, pp. 367–377, http://dx.doi.org/10.1109/ICSME.2016.65.

[14] M.B. Zanjani, S. Member, Automatically recommending peer reviewers in modern code review, IEEE Trans. Softw. Eng. 42 (6) (2016) 530–543, http://dx.doi.org/10.1109/TSE.2015.2500238.

[15] E. Sülün, E. Tüzün, U. Dogrusöz, Reviewer recommendation using software artifact traceability graphs, in: PROMISE'19: Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering, 2019, pp. 66–75, http://dx.doi.org/10.1145/3345629.3345637.

[16] J. Jiang, J.-H. He, X.-Y. Chen, CoreDevRec: Automatic core member recommendation for contribution evaluation, J. Comput. Sci. Tech. 30 (5) (2015) 998–1016, http://dx.doi.org/10.1007/s11390-015-1577-3.

[17] Z. Xia, H. Sun, J. Jiang, X. Wang, X. Liu, A hybrid approach to code reviewer recommendation with collaborative filtering, in: 2017 6th International Workshop on Software Mining, 2017, pp. 24–31.

[18] E. Dogan, E. Tuzun, K.A. Tecimer, H.A. Guvenir, Investigating the validity of ground truth in code reviewer recommendation studies, in: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2019, pp. 1–6, http://dx.doi.org/10.1109/ESEM.2019.8870190.

[19] V. Kovalenko, N. Tintarev, E. Pasynkov, C. Bird, A. Bacchelli, Does reviewer recommendation help developers? IEEE Trans. Softw. Eng. (2018) 1, http://dx.doi.org/10.1109/TSE.2018.2868367.

[20] D. Cubranic, G.C. Murphy, Automatic bug triage using text categorization, in: SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering, KSI Press, 2004, pp. 92–97.

[21] L. Jonsson, M. Borg, D. Broman, K. Sandahl, S. Eldh, P. Runeson, Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts 21, 2016, http://dx.doi.org/10.1007/s10664-015-9401-9.

[22] H. Hu, H. Zhang, J. Xuan, W. Sun, Effective bug triage based on historical bug-fix information, in: 2014 IEEE 25th International Symposium on Software Reliability Engineering, 2014, pp. 122–132, http://dx.doi.org/10.1109/ISSRE.2014.17.

[23] H. Naguib, N. Narayan, B. Brügge, D. Helal, Bug report assignee recommendation using activity profiles, in: 2013 10th Working Conference on Mining Software Repositories (MSR), 2013, pp. 22–30, http://dx.doi.org/10.1109/MSR.2013.6623999.

[24] A. Søgaard, B. Plank, D. Hovy, Selection Bias, Label Bias, and Bias in Ground Truth, in: Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Tutorial Abstracts, 2014.

[25] G.F. Cabrera, C.J. Miller, J. Schneider, Systematic labeling bias: De-biasing where everyone is wrong, in: Proceedings - International Conference on Pattern Recognition, 2014, http://dx.doi.org/10.1109/ICPR.2014.756.

[26] E. Sulun, E. Tuzun, U. Dogrusoz, Rstrace+: Reviewer suggestion using software artifact traceability graphs, Inf. Softw. Technol. 130 (2021) 106455, http://dx.doi.org/10.1016/j.infsof.2020.106455, URL: https://www.sciencedirect.com/science/article/pii/S0950584920300021.

[27] M. Fejzer, P. Przymus, K. Stencel, Profile based recommendation of code reviewers, J. Intell. Inf. Syst. (2018) http://dx.doi.org/10.1007/s10844-017-0484-1.

[28] S.-R. Lee, M.-J. Heo, C.-G. Lee, M. Kim, G. Jeong, Applying deep learning based automatic bug triager to industrial projects., in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 926–931, URL: https://search.ebscohost.com/login.aspx?direct=true&db=edscma&AN=edscma.3117776&site=eds-live.

[29] S.F.A. Zaidi, F.M. Awan, M. Lee, H. Woo, C.-G. Lee, Applying convolutional neural networks with different word representation techniques to recommend bug fixers, IEEE Access 8 (2020) 213729–213747, http://dx.doi.org/10.1109/ACCESS.2020.3040065.

[30] H.A. Cetin, E. Dogan, E. Tuzun, A review of code reviewer recommendation studies: Challenges and future directions, Sci. Comput. Program. 208 (2021) 102652, http://dx.doi.org/10.1016/j.scico.2021.102652, URL: https://www.sciencedirect.com/science/article/pii/S0167642321000459.

[31] S. Rebai, A. Amich, S. Molaei, M. Kessentini, R. Kazman, Multi-objective code reviewer recommendations: Balancing expertise, availability and collaborations, Autom. Softw. Engg. 27 (3–4) (2020) 301–328, http://dx.doi.org/10.1007/s10515-020-00275-6.

[32] A. Strand, M. Gunnarson, R. Britto, M. Usman, Using a context-aware approach to recommend code reviewers: Findings from an industrial case study, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice, in: ICSE-SEIP '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 1–10, http://dx.doi.org/10.1145/3377813.3381365.

[33] J. Lipcak, B. Rossi, A large-scale study on source code reviewer recommendation, in: 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2018), 2018.

[34] T. Zhang, J. Chen, G. Yang, B. Lee, X. Luo, Towards more accurate severity prediction and fixer recommendation of software bugs, J. Syst. Softw. 117 (2016) 166–184, URL: https://search.ebscohost.com/login.aspx?direct=true&db=edselp&AN=S0164121216000765&site=eds-live.

[35] X. Xia, D. Lo, Y. Ding, J.M. Al-Kofahi, T.N. Nguyen, X. Wang, Improving automated bug triaging with specialized topic model, IEEE Trans. Softw. Eng. 43 (3) (2017) 272–297, http://dx.doi.org/10.1109/TSE.2016.2576454.

[36] S. Mani, A. Sankaran, R. Aralikatte, DeepTriage: Exploring the Effectiveness of Deep Learning for Bug Triaging, Association for Computing Machinery, New York, NY, USA, 2019, http://dx.doi.org/10.1145/3297001.3297023.

[37] P. Bhattacharya, I. Neamtiu, C.R. Shelton, Automated, highly-accurate, bug assignment using machine learning and tossing graphs, J. Syst. Softw. 85 (10) (2012) 2275–2292, http://dx.doi.org/10.1016/j.jss.2012.04.053, URL: https://www.sciencedirect.com/science/article/pii/S0164121212001240. Automated Software Evolution.

[38] D. Yang, Y. Lei, X. Mao, D. Lo, H. Xie, M. Yan, Is the ground truth really accurate? Dataset purification for automated program repair, in: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2021, pp. 96–107, http://dx.doi.org/10.1109/SANER50967.2021.00018.

[39] K.A. Tecimer, E. Tüzün, H. Dibeklioglu, H. Erdogmus, Detection and elimination of systematic labeling bias in code reviewer recommendation systems, in: Evaluation and Assessment in Software Engineering, in: EASE 2021, Association for Computing Machinery, New York, NY, USA, 2021, pp. 181–190, http://dx.doi.org/10.1145/3463274.3463336.

[40] R. Mohanani, I. Salman, B. Turhan, P. Rodríguez, P. Ralph, Cognitive biases in software engineering: A systematic mapping study, IEEE Trans. Softw. Eng. 46 (12) (2020) 1318–1339, http://dx.doi.org/10.1109/TSE.2018.2877759.

[41] P. Ralph, Toward a theory of debiasing software development, in: Lecture Notes in Business Information Processing, vol. 93, 2010, pp. 92–105, http://dx.doi.org/10.1007/978-3-642-25676-98.

[42] W. Stacy, J. Macmillan, Cognitive bias in software engineering, Commun. ACM 38 (6) (1995) 57–63.

[43] E. Smith, A.T. Bahill, Attribute substitution in systems engineering, Syst. Eng. 13 (2009) 130–148, http://dx.doi.org/10.1002/sys.20138.

[44] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, P. Devanbu, Fair and balanced? Bias in bug-fix datasets, 2009, pp. 121–130, http://dx.doi.org/10.1145/1595696.1595716.

[45] T.H.D. Nguyen, B. Adams, A.E. Hassan, A case study of bias in bug-fix datasets, 2010, http://dx.doi.org/10.1109/WCRE.2010.37.

[46] K. Herzig, A. Zeller, The impact of tangled code changes, in: Proceedings of the 10th Working Conference on Mining Software Repositories, in: MSR '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 121–130, URL: http://dl.acm.org/citation.cfm?id=2487085.2487113.

[47] A. Ahluwalia, D. Falessi, M.D. Penta, Snoring : a noise in defect prediction datasets, in: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), IEEE, 2019, pp. 63–67, http://dx.doi.org/10.1109/MSR.2019.00019.

[48] T.-H. Chen, M. Nagappan, E. Shihab, A.E. Hassan, An empirical study of dormant bugs, in: Proceedings of the 11th Working Conference on Mining Software Repositories, in: MSR 2014, Association for Computing Machinery, New York, NY, USA, 2014, pp. 82–91, http://dx.doi.org/10.1145/2597073.2597108.

[49] M. Rath, P. Mäder, The SEOSS 33 dataset — Requirements, bug reports, code history, and trace links for entire projects, Data Brief (2019) http://dx.doi.org/10.1016/j.dib.2019.104005.

[50] S. Dueñas, V. Cosentino, G. Robles, J.M. Gonzalez-Barahona, Perceval: Software project data at your will, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, 2018, pp. 1–4.

[51] K.A. Qamar, E. Sulun, E. Tuzun, Towards a taxonomy of bug tracking process smells: A quantitative analysis, in: 2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2021, pp. 138–147, http://dx.doi.org/10.1109/SEAA53835.2021.00026.

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in python, J. Mach. Learn. Res. 12 (null) (2011) 2825–2830.

[53] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, in: Y. Bengio, Y. LeCun (Eds.), 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings, 2013, URL: http://arxiv.org/abs/1301.3781.

[54] P. Thongtanunam, R.G. Kula, A.E.C. Cruz, N. Yoshida, H. Iida, Improving code review effectiveness through reviewer recommendations, in: Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering, in: CHASE 2014, Association for Computing Machinery, New York, NY, USA, 2014, pp. 119–122, http://dx.doi.org/10.1145/2593702.2593705.

[55] E.M. Voorhees, D.M. Tice, The TREC-8 question answering track, in: Proceedings of the Second International Conference on Language Resources and Evaluation (LREC'00), European Language Resources Association (ELRA), Athens, Greece, 2000, URL: http://www.lrec-conf.org/proceedings/lrec2000/pdf/26.pdf.

[56] H. Song, M. Kim, D. Park, Y. Shin, J.-G. Lee, Learning from noisy labels with deep neural networks: A survey, 2020, arXiv preprint arXiv:2007.08199.

[57] X. Xia, D. Lo, E. Shihab, X. Wang, B. Zhou, Automatic, high accuracy prediction of reopened bugs, Autom. Softw. Eng. (2014) http://dx.doi.org/10.1007/s10515-014-0162-2.

[58] A. Fernández, S. García, M. Galar, R. Prati, B. Krawczyk, F. Herrera, Learning from imbalanced data sets, 2018, http://dx.doi.org/10.1007/978-3-319-98074-4.

[59] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empir. Softw. Eng. 14 (2) (2008) 131, http://dx.doi.org/10.1007/s10664-008-9102-8.