Detection and Elimination of Systematic Labeling Bias in Code Reviewer Recommendation Systems

K. Ayberk Tecimer
Technical University of Munich
Munich, Germany
ayberk.tecimer@tum.de

Hamdi Dibeklioğlu
Bilkent University
Ankara, Turkey
dibeklioglu@cs.bilkent.edu.tr

Eray Tüzün
Bilkent University
Ankara, Turkey
eraytuzun@cs.bilkent.edu.tr

Hakan Erdogmus
Carnegie Mellon University
Pittsburgh, USA
hakane@andrew.cmu.edu

ABSTRACT
Reviewer selection in modern code review is crucial for effective code reviews. Several techniques exist for recommending reviewers appropriate for a given pull request (PR). Most code reviewer recommendation techniques in the literature build and evaluate their models based on datasets collected from real projects using open-source or industrial practices. The techniques invariably presume that these datasets reliably represent the “ground truth.”

In the context of a classification problem, ground truth refers to the objectively correct labels of a class used to build models from a dataset or evaluate a model’s performance. In a project dataset used to build a code reviewer recommendation system, the recommended code reviewer picked for a PR is usually assumed to be the best code reviewer for that PR. However, in practice, the recommended code reviewer may not be the best possible code reviewer, or even a qualified one. Recent code reviewer recommendation studies suggest that the datasets used tend to suffer from systematic labeling bias, making the ground truth unreliable. Therefore, models and recommendation systems built on such datasets may perform poorly in real practice.

In this study, we introduce a novel approach to automatically detect and eliminate systematic labeling bias in code reviewer recommendation systems. The bias that we remove results from selecting reviewers that do not ensure a permanently successful fix for a bug-related PR. To demonstrate the effectiveness of our approach, we evaluated it on two open-source project datasets—HIVE and QT Creator—and with five code reviewer recommendation techniques—Profile-Based, RTrace, Naive Bayes, k-NN, and Decision Tree. Our debiasing approach appears promising since it improved the Mean Reciprocal Rank (MRR) of the evaluated techniques up to 26% in the datasets used.

CCS CONCEPTS
• Software and its engineering → Software development process management; Collaboration in software development.

KEYWORDS
modern code review, ground truth, labeling bias elimination, systematic labeling bias, data cleaning, code review recommendation

1 INTRODUCTION
The code review process is an important step in the software development lifecycle. Effective code reviews increase internal quality and reduce defect rates [25]. To increase the effectiveness of code reviews, reviewers should be selected carefully. Several Code Reviewer Recommendation (CRR) techniques exist in the literature [4, 15, 17, 21, 29, 31, 33–35]. These CRR techniques use different strategies, but they invariably either build or evaluate their models based on datasets gathered from industrial or 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 captured by a pull request (PR), in a dataset is the best possible reviewer (the assignment is done by carefully evaluating candidate reviewers and selecting the best one is truly the best qualified for that PR.) However, in practice, the selected code reviewer may not be the most qualified, or even sufficiently qualified to review the submitted PR [9]. In several scenarios, reviewer assignments tend to be based on non-technical factors, which may invalidate the central assumption of the models built [9].

For instance, according to a study on code reviewer practices at Microsoft [16], reviewers are assigned to PRs according to their availability and social relationship with the person who makes the reviewer assignments. Similarly, according to Dogan et al. [9], availability is an important factor for reviewer assignments and is frequently substituted for technical or competency factors. In other words, recommendation labels in datasets may frequently
be wrong. Therefore datasets originating from real practice can negatively affect the accuracy and reliability of the CRR techniques that rely on them.

In machine learning, the kind of labeling error that exists in CRR 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 obtained in the training samples since there are several factors, including subjective ones, to be considered. Although the labels are not perfect, they are still considered as the ground truth. While the amount of problematic labels in the ground truth may be relatively small or the inconsistency in the class labels may be negligible, in some cases the ground truth may include systematic problems that can cause the models not be able to converge, learn generalizable patterns, or, as in the CRR case, not to be as effective as they could be in practice. Such issues are generally due to basic/naïve assumptions in the labeling process [27] or intrinsic properties of the observed data [7]: this is what systematic labeling bias in general refers to.

With the goal to prevent these kinds of labeling problems in the ground truth of CRR datasets, we formulate two research questions:

- RQ1: How can we eliminate systematic labeling bias in CRR datasets?
- RQ2: How does systematic labeling bias elimination affect the performance of CRR techniques?

For RQ1, we explore possible solutions and introduce a new approach to detect and eliminate potentially “incorrect” reviewer labels in CRR datasets. For RQ2, we measure the effects of our proposed approach by comparing before and after accuracy rates of five CRR techniques: Naïve Bayes, k-NN, Decision Tree, RSTrace and Profile based.

Section 2 provides the background information on the modern code review practice, CRR approaches, ground truth problems in software engineering, and cognitive bias in software engineering. Section 3 defines a success criterion for code reviews for identified incorrect assignments. Furthermore, Section 3 focuses on code reviews associated with PRs and bugs and introduces our debiasing (data cleaning) approach. Section 4 describes our experiments including the datasets, preprocessing steps, and experimental setup, and presents the results. Section 5 answers the research questions and discusses the limitations of our approach. Finally, Section 6 summarizes the contribution and discusses future work.

2 BACKGROUND AND RELATED WORK

Code review is a central quality practice and an important part of the software development lifecycle. Traditional, Fagan-style code review [10] is a formal, manual, synchronous, and well-documented process performed by a carefully assigned group on selected parts of the codebase. In contrast, modern code review is informal, tool-based, asynchronous, and focuses on reviewing only latest changes [2, 25]. In the last two decades, modern code review has become dominant in both commercial projects [6, 25] and open-source ones [3].

In the following, we provide a summary of code review recommendation techniques, ground truth problems and cognitive bias examples in software engineering.

2.1 Code Review 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. [36].

Optimization-based approaches. Balachandran [4] proposed a heuristic that analyzes the change history to find suitable reviewers. They reach 60% - 92% recommendation accuracy, which is better than a comparable code reviewer recommendation approach based on file change history. Lee et al. [17] 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. [31]; later Xia et al. [33] extended this technique with text mining to leverage additional information in recommendations. Ouni et al. [21] 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 accurately recommends code reviewers with up to 59% of precision and 74% of recall. Zanjani et al. [35] 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 the prior code reviewer recommendation approaches. Sun et al. [30] proposed a graph-based technique using traceability relations between PRs, source code files, and bugs to recommend code reviewers for a given pull request.

Learning-based approaches. Jiang et al. [15] proposed a technique that builds a model with Support Vector Machines (SVM) to make reviewer recommendations in OSS projects. The authors evaluate their technique on 18,651 pull requests of five popular projects in GitHub. They indicate that their technique achieves accuracy from 72.9% to 93.5% for Top-3 recommendation. Xia et al. [34] presented a hybrid approach in which they combined latent-factor models and neighborhood methods. Their results demonstrate that the proposed approach performs better than comparing methods for all Top-k recommendations.

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 [18, 21, 34, 35] acknowledge this problem explicitly as a limitation. For instance, Ouni et al. [21] 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 [18] 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 Ground Truth Problems in Software Engineering

Outside CRR [9], ground truth problems exist in data-analytics solutions that support other common predication and recommendation tasks in software engineering.

Bird et al. [5] 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. [20] 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 both linkage and tagging biases even in datasets thought to be near accurate.

Herzig et al. [14] 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. [1] 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. [8] analyzed dormant bugs in 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 ground truth.

2.3 Cognitive Bias Examples in Software Engineering

A cognitive bias is a type of systematic error in decision making that causes suboptimal outcomes based on established beliefs and misguided intuition. Our work 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. [19] provide many examples of cognitive biases in software engineering in their literature review and advocate theirmitigation as central to improving the quality of decision making. Ralph [22] attributes the persistence of high software project failure rates despite the many advances in software technologies and processes to cognitive biases. Stacy and Macmillian [28] state that cognitive biases have an adverse effect on software developers’ thought processes. They advise preferring empirical investigation to intuition and seeking disconfirmatory information to reduce cognitive biases. Smith and Bahill [26] similarly argue 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 suggest 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 datasets caused by past reviewer assignments based on convenience and social factors rather than factors that are directly related to the suitability of a reviewer for the specific review task.

3 IMPROVING CRR ACCURACY IN PAST DATA

In the remainder of the paper, we assume that a code review task is encapsulated in a pull request (PR). The purpose of code review is to merge the PR successfully into the codebase after having addressed all of the reviewers comments, and close the PR, hopefully indefinitely. The task of the code reviewer is to ensure this successful resolution. According to Google best practices for code reviews [13], “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.” This definition of an appropriate, or successful, code reviewer, aligns with that of a successful PR as one that avoids downstream rework related to the changes contained in the PR after the PR has been closed. However the success criterion is subjective as is, and not easily measurable. This poses a problem since our goal is to eliminate systematic labeling bias in CRR datasets: we need a way to identify whether a label, in this case the assigned reviewer, is right or wrong. The label will be wrong if the PR is unsuccessful, in which case we need to be able to remove the bias either by correcting the label (assigned reviewer), removing the data point corresponding to that PR, or generating new data with checks that increase label accuracy.

3.1 Manual Correction

If the success measure is qualitative, we can only use manual methods to remove it. For example, additional expert checks may be put in place to tag low-quality reviews when a PR is closed. Or multiple independent reviewers can be assigned to a single PR, and an expert again can examine and tag low-quality reviews. The tagged reviews can then be excluded from the generated reviewer assignment data. This can also be done post hoc: the reviewer assignment data can be cleaned by an expert after the fact using similar quality checks. However, these approaches would not scale up well because they can be too costly or too impractical, or both. Besides, manual expert checks can still be error-prone, and we must still assume the expert performing the checks and verifying the labels is unbiased; if the expert is biased, we create a circular situation in which we attempt to address one kind of bias by a method that introduces another kind of bias.
3.2 Characterization of a Successful Code Review

When PRs are associated with coding tasks or defects in a bug tracking system, an objective success measure can be defined: if the associated bug 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 master branch to close the PR. The code review task itself associated with the PR is successful if the PR is successful. Similarly, the reviewers performing the code review for the PR are themselves successful if the PR they helped close is successful.

While this success measure is objective, it still has a few caveats. First, it cannot be immediately determined whether a PR is successful: we can only decide the status after the closing of the PR. The second one is: it relies on the assumption that reopening the underlying bug after the PR has been closed can only be due to the original PR not having been successful, and not due to, for example, something outside the scope of the PR having changed causing a ripple effect (a future commit refactoring a dependency.) The third caveat is that it assumes perfect bug identification: a new bug is created only when the reason for the bug cannot reliably traced to an existing bug 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 techniques with this success measure.

3.3 The Debiasing Approach

The objective success measure defined in the previous section provides us with an easy way to identify incorrect labels in CRR datasets and remove them. The success measure requires linkages between PRs and bugs logged in a bug tracking system. Almost all datasets that have the required linkages focus on bug-related PRs. The CRR techniques on which we have tested our approach do the same. So we limit ourselves to bugs and PRs associated with bugs.

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. 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 unsuccessful. When the reviewers finally approve the fix, the developer merges the PR to the master and the bug’s status is changed to Closed. The merged PR is eventually deployed. However, further testing, field testing by end users, or ongoing development work may at one point reveal that the bug was not 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 will discover in retrospect that the original PR was in fact not successful, and the reviewer assignment associated with this PR becomes a candidate for removal.

Therefore our debiasing approach is simple and based on the removal of unsuccessful PRs. Given a CRR dataset consisting of a set of PRs, reviewer assignments for each PR, a bug associated with each PR, the status history of each bug in the bug tracking system, we check for a PR whether the associated bug was reopened after it was closed following the merge. If it was, we consider this PR to be unsuccessful post hoc and remove the associated data point from the dataset along with the reviewer assignment.

Figure 1: The lifecycle of a bug and where it interacts with the PR process

4 EXPERIMENTS AND RESULTS

4.1 Dataset Description and Preprocessing

Having defined the debiasing method, we evaluate it on two different datasets. These datasets belong to projects from two sources: Qt, a company that develops cross-platform software, and from Apache, a widely used open-source cross-platform software foundation. 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 and bug history until December 2019. For HIVE, we used the version provided by SEOSS 33, 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, which allows fetching datasets from both GitHub and Jira. Most of the PRs in the two datasets are associated with Jira bug ID (in HIVE 96.34%, and in QT-Creator 73.18%). This allowed us tracing PRs to Jira bugs.

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 found and merged automatically if different names correspond to the same email address. In the third step, we checked whether any 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 5,927. 406 of these pull requests were unsuccessful, corresponding to a failure/success ratio of 7%. The HIVE dataset had 3,621 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.

### 4.2 Evaluation Setup

To evaluate the reliability and usefulness of the debiasing approach, we selected five different CRR techniques from the literature, namely, Profile-based [11], RSTrace [29], Naïve Bayes, k-NN (5-NN), and Decision Tree. Initially, we wanted to apply the approach to all CRR techniques discussed under the section on background and related work. However, only a few of the CRR techniques [11, 29] provide source code or pseudocode. Therefore, we selected those that we could actually run or 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. For Naïve-Bayes, 5-NN, and Decision Tree, we used the implementations provided by the Scikit-learn library.6

For the three machine learning techniques (Naïve Bayes, k-NN, and Decision Tree), we used file-paths in PRs as features. To convert these file-paths to numeric values (for using in classification), we applied the vectorizers (CountVectorizer and TfidfVectorizer) from the Scikit-learn library. Hyperparameters of the learning-based models were optimized within a set of considered values. The considered values are given in Table 2.

### 4.3 Performance Measures

The accuracy of the CRR techniques selected were assessed by widely two used measures: Top-k accuracy (namely, Top-1, Top-3 and Top-5) and Mean Reciprocal Rank (MRR). Top-k accuracy computes the ratio of test cases that have the correct label within the top k predictions to all cases. MRR is the inverse of the rank of the first correct answer.

6https://scikit-learn.org/

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

\[
\frac{S_{\text{after}}}{S_{\text{before}}} = 1
\]

where, \(S_{\text{before}}\) is the performance of a CRR technique before we applied debiasing to its training dataset and \(S_{\text{after}}\) is the performance after we applied debiasing to the same dataset.

### 4.4 Balancing

The selected CRR techniques were trained and tested with the HIVE and QT Creator datasets. Both datasets had quite low unsuccessful PR rate (5% to 7%). However, according to the literature [32], unsuccessful PRs are significantly more pervasive than they appear to be in our datasets. The actual reopened bug ratios in some popular open-source projects are much higher: for example, in Eclipse it was found to be 16.1% and in OpenOffice as high as 26.31%. A lower than actual ratio commonly stems from the practice of opening a new bug for convenience because searching for the original bug, identifying it, and reopening it may require effort. Developers also may not remember or know about the source bug due to turnover or time lapse, and inadvertently mistake a recurring one for a brand new bug.

The problem with a low unsuccessful PR rate in the dataset due to missed reopened bugs is that debiasing through the removal of the corresponding data points will inevitably yield only marginal improvements in performance. We are interested in assessing how much improvement can be achieved by debiasing if the training dataset’s ratio was closer to the actual ratios observed in real practice. Therefore, we under-sampled the data [12] by randomly removing successful PRs until the unsuccessful ratio is 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 quadruples the unsuccessful PR ratios, bringing them closer to the more commonly observed values, and making the dataset more balanced for training.

### 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 Debias (Db) dataset. At the first step, a CRR 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 with bad labels.

Our datasets contain too few datapoints that exhibit labeling bias, so we expect the improvement to be marginal. At the second step, we want to see how much the effects are amplified when systematic labeling bias is as pervasive as it is reported in the literature. We balance the PRs to increase 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 CRR technique. The improvement in performance should now be more increased.

Finally, we perform an extra validation step 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 given CRR 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.

### 4.6 Testing Strategies for the CRR Techniques

While testing both categories of techniques, we preserved the chronological order of the data since we should not attempt to predict past instances using future instances. Therefore, we made sure 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 (Naive Bayes, k-NN [5-NN], and Decision Tree), we performed sliding window testing. We divided our dataset into 10 folds, choose a test fold, and trained our models using all the folds chronologically located before the test fold. In each iteration, we slid the window to the next fold and repeated the same procedure.

### 4.7 Results: Effect of Debiasing

In order to investigate the effect of removing unsuccessful PRs from the datasets, we evaluated the accuracy of the five CRR techniques on QT Creator and HIVE datasets before and after the debiasing. Notice that after debiasing, the datasets contain only successful PRs. The CRR techniques were trained with the same parameters to make the comparisons valid.

Table 3 summarizes the Top-3 and Top-5 correct classification rates 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 Naive 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 Fig. 5 and Fig. 6, respectively. Higher accuracy rates were obtained for higher “k” values in Top-k overall. For any k value, we did not observe a considerable improvement in the accuracy of optimization-based approaches 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 overall performance. Fig. 6 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 HIVE is 25% and for Naive Bayes technique on QT Creator dataset is 26%. These improvements are higher than the improvements observed with the Top-k measures.

### 4.8 Does Debiasing Actually Work or Is It Just Coincidence?

To illustrate that improvements observed with debiasing is no accidental, we performed an additional validation step:

- Remove PRs randomly from the datasets and evaluate the accuracy of the CRR techniques 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 for 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 averaged.
Figure 2: The evaluation process using different versions of the preprocessed datasets

Figure 3: Percentage of unsuccessful PRs detected at each day following a successful merge due to reopening of the underlying bug (HIVE dataset)

- Remove only successful PRs from the datasets (thus introducing a bias against successful PRs) and evaluate the accuracy of the CRR techniques before and after debiasing. This process compares the dataset version RBi (Reduced-Biased) in Fig. 2 to the version DbBa. Again the random reduction is repeated 100 times and performance results averaged.

Table 4 summarizes the results of this step for the MRR measure. The results are expected. All CRR techniques perform best without unsuccessful PRs (DbBa) and worst with randomly removed successful PRs (RUi). In all cases, the CRR techniques invariably perform 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 give better results.

4.9 How Many Unsuccessful PRs Could be Superfluous?

In Section 3.2, we discussed a number of caveats for the PR success measure adopted. We now focus on the second caveat to assess to what extent reopened bugs 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 bug was a false positive due to changes to the codebase unrelated to the original bug (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 analyze the elapsed time between Closed and Reopened transitions of each reopened bug in the datasets. If the elapsed time is small, e.g., less than one day, resurfacing of the bug is unlikely to be related to external circumstances impossible to have been detected by the PR reviewers. We also look 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 is high after a certain time has lapsed, their
likelihood of causing the bug to resurface is low, hence the PR was likely genuinely unsuccessful.

Fig. 3 shows that in the Hive dataset, 45% of the bugs are reopened on the same day of the PR and 80% of the bugs are reopened within 24 days. Fig. 4 shows that 80% of the files involved in an unsuccessful PR remain untouched 19 days after the closure of the underlying bug. The results are similar on the second dataset. Because most bugs are reopened in the first few days of their closure and most files involved in an unsuccessful PR remain untouched during the initial days after closure, we believe mislabeling unsuccessful PRs, although possible in rare circumstance, is unlikely to be pervasive enough to compromise the PR success measure.

To validate whether reopened PRs indeed had poor reviews, we performed a quality analysis of the PRs proposed in a random 10% sample (40 data points). You can find this analysis in a supplement posted to Figshare. 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 (LGTM/missing comments) (21), Overruled (author indicated reviewer had misjudged the changes) (3), and Poor-Effort (self-admittance of a sub-standard/rushed review) (6).

5 DISCUSSION

5.1 Research Questions

RQ1: How can we eliminate the systematic labeling bias in CRR datasets?

Since manual methods are not cost-efficient, can still be error-prone, and do not scale up well, we looked for an automatable method based on an objective success measure that leverages linkages between PR data and bug data. 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 code changes. The debiasing method we propose simply removes unsuccessful samples from the PR data to eliminate possible biases in past reviewer selections. The CRR techniques could then use this debiased data as their ground truth to build their models and improve their performance.

RQ2: How does systematic labeling bias elimination affect the performance of CRR techniques?

We applied the automatic debiasing methods to a diverse set of five CRR techniques using two open-source datasets. We observed that, provided the data had sufficiently high-rates of badly labeled samples reported in the literature, the performance of the CRR techniques in general improved after debiasing. The highest improvement was observed with learning-based CRR techniques. The improvements in the optimization-based CRR techniques tested were marginal.

The reason behind the difference of improvement between optimization-based vs. learning-based techniques is that the reviewer recommendations process of the optimization-based techniques tested do not as heavily depends on learning from the past data as those of learning-based techniques. Additionally, not every sample may be as equally valuable since the optimization criterion may inadvertantly already discount badly labeled samples. Therefore, a debiasing approach focused on removing badly labeled samples may not make much difference in such techniques.

We conclude that the proposed debiasing approach improves the quality of the ground truth and is worthwhile for learning-based CRR techniques.

5.2 Threats to Validity

Our method of identifying a badly labeled PR is subject to a construction threat [24]. It may not be possible to catch all reopened bugs: some may be completely missed, others may yet to be reopened and not captured in the dataset. Leaving these false negatives in the dataset would reduce the efficacy of debiasing. Conversely, there may be false positives: PRs identified as unsuccessful due to a reopened bug may have actually been successful. We discussed some possible reasons for such cases in Section 4.9, and by examining two indicators in the datasets, concluded that these cases should be rare.

Our original datasets had low unsuccessful PR rates, due to the fact that some unsuccessful PRs are likely to be mislabeled as successful because the recurrence of the associated bugs was missed in the bug tracking system (we discussed possible reasons for this.) At these low rates, the introduced bias is not significant, and removing it would not yield much benefit. We precisely observed 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 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 posit that any observed improvements are conditional on a dataset having sufficient systematic labeling bias.

Threats to external validity are concerned with generalizability [24]. We evaluated our approach on two open-source datasets and five different CRR techniques. We believe the CRR techniques used are a reasonable representation of common approaches. However we acknowledge the limitations of using two datasets. We could not identify further datasets that both contain sufficient samples and integrate PR information with bug tracking.

Open-source projects typically have high turnover rates. Unlike in closed-source projects, many contributors and reviewers become inactive over time, and others joint the project. Therefore, our observations may not apply to closed-source projects.

To mitigate internal threats related to the implementation of the CRR techniques and the data extraction methods, we provide the source code of all techniques used in the evaluation as well as the both datasets in Figshare.

6 CONCLUSION AND FUTURE WORK

Good code reviewer selection is central to effective code reviews. CRR techniques attempt to automate the code reviewer selection problem, but they mostly 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

1 https://figshare.com/s/1b9e55737d9f2c31a7a

2 https://figshare.com/s/1b9e55737d9f2c31a7a
attribute for a competence attribute, in reviewer assignments. When the code reviews are performed in the context of PRs, and the dataset links PR data with bug tracking data, it is possible to identify PR requests that fail to achieve their goal: a PR, and the developer assigned to reviewing it, can be deemed unsuccessful when the bug associated the PR is reopened later. Our experiments showed that when such failed cases are pervasive enough—in the 20–28% range consistent with reported rates—removing such data points from the dataset in general improves the performance of CRR techniques. Although the improvement was very marginal for optimization-based CRR techniques tested, it was large for learning-based CRR techniques (up to 26% for Naïve Bayes).

Our work has implications for both practitioners and researchers. Researching can apply our proposed debiasing approach while cleaning their training data to improve the accuracy of their CRR models. Recommendation tools built on these models would then inherit these improvements. The debiasing approach could also be useful for flagging potentially ineffective reviews to improve the code review practices in an organization.

Although initial results look promising, we still need to test our approach on other datasets and datasets collected from commercial systems. We are in the process of looking for suitable candidates. It would make sense for the future work to focus on learning-based CRR techniques since the real returns are to found in that space. If expanded evaluations prove the debiasing method to be widely effective, we plan to provide tool support for automated debiasing.

REFERENCES


