Duplicate Bug Report Detection

Tarun Chhabra
Computer Science Department
North Carolina State University
Raleigh, NC, 27606, USA
tchhabr@ncsu.edu

Devika Desai
Computer Science Department
North Carolina State University
Raleigh, NC, 27606, USA
dndesai@ncsu.edu

Sudipto Biswas
Computer Science Department
North Carolina State University
Raleigh, NC, 27606, USA
sbiswas4@ncsu.edu

ABSTRACT

There are many users interacting with a system and reporting issues concerned with it in terms of a bug report. Bug reports are then used to guide software corrective maintenance activities so that it can result in more reliable software systems. Bug repositories are maintained as a collection of bug reports. Despite the advantages of a bug report system, it does cause some challenges. As bug reporting process is often ad-hoc, often the same bugs are reported by different users, or a different bug caused by the same potential software defect result in duplicate bug reports. A number of studies have attempted to address this issue by automating bug-report deduplication. The following review provides an overview of all the techniques used for duplicate bug detection before the concept of detection by information retrieval [1] and how the field has changed after that. A number of past studies have proposed a number of automated approaches to detect duplicate bug reports. We comment on how these approaches can be improved by integrating two or more techniques or considering even more possible factors for accurate detections.

Keywords

Duplicate bugs; triaging; bug reports; de-duplication; similar reports; NLP

1. INTRODUCTION

Due to system complexity and inadequate testing, many software systems are often released with defects or have some unknown bugs. Bugs occur for a variety of reasons, ranging from ill-defined specifications to carelessness, to programmers misunderstanding of the problem, technical issues, nonfunctional qualities, corner cases, etc. To overcome such situations developers often need proper feedback on the bugs that are present in the systems. For this, they allow users to report such bugs using bug report systems such as Bugzilla, Jira or other propriety systems. With such systems, end users and testers could report bugs that they encounter and developers could triage, track, and comment on the various bugs that are reported. Bug reporting is standard practice in both open source software development and closed source software development.

For several reasons, such as lack of motivation of users and defects in the search engine of the bug tracking systems, the users of software systems may report bugs that are already present in the bug tracking system. These bug reports are called “duplicates”. The word duplicate may also represent the bug reports referring to different bugs in the system that are caused by the same software defect.

To automate the detection of duplicate bug reports, several approaches have been introduced so far in the papers we studied. Early approaches have applied information retrieval (IR) to this problem with Vector Space Model (VSM) in which a bug report is modeled as a vector of textual features computed via Term Frequency-Inverse Document Frequency (Tf-Idf) term weighting measurement. To improve the detection accuracy, natural language processing (NLP) has been combined with those IR methods. Execution trace information on the reported bugs in the bug reports is also used in combination with NLP. However, execution traces might not be available in all bug reports. Another predominant approach to this problem is machine learning (ML). Jalbert et al [22] use a binary classifier model and apply a linear regression over textual features of bug reports computed from their terms’ frequencies. To train an SVM classifier, all pairs of duplicate bug reports are formed and considered as the positive samples and all other pairs of non-duplicate bug reports

are used as the negative ones. We gathered from this paper that the key limitation of ML approaches is their low efficiency. The recent work by Sun has shown that REP, an advanced IR approach, outperformed state-of-the–art ML approaches in term of both accuracy and time efficiency. It is customized from BM25F to take into account the long bug reports and the meta-data such as the reported product, component, and version. BM25F is a textual similarity classification algorithm for structured text retrieval. This algorithm comprises of several fields such as, headlines, main text etc. These fields are categorized in different possible degrees of importance, term relevance and normalization. The key assumption in REP is based on a high textual similarity between duplicate bug reports. REP is the novel technique introduced in [2] to measure the similarity between two bug reports. It uses not only the similarity of textual content in summary and description fields available in a bug report including, but also the similarity of non-textual fields such as product, component, version etc. However, we understand that in practice, it is popular that the bug reports can be filed by multiple reporters who could describe the same technical issues in different phenomena via different terms. With different input data, usage environments and scenarios, an erroneous behavior might be exposed as different phenomena (e.g. different outputs, traces, or screen view). Moreover, different reporters might use different terminologies and styles, or write about different phenomena to describe the same issues. Thus, duplicate bug reports might not be very textually similar. In those cases, REP does not detect them well.
Through our readings we came to know of another approach: that is DBTM, a duplicate bug report detection model that takes advantage of not only IR-based features but also topic based features from the novel-topic model, which is designed to address textual dissimilarity between duplicate reports.
Bug fixing is important in producing high-quality software. Bug fixing can be carried out in both development and post-release time. In both the cases, the developers carry out necessary testing and find the incorrect behaviors that do not conform with their expectations and software requirements.

2. MOTIVATION

The motivation behind our chosen topic and the study of its related papers is that even though Duplicate bug report detection has been actively researched and many techniques have been proposed to solve it, each technique is not fully successful to detect the duplication accurately. By studying different papers related to this topic we came across how different techniques can be combined to achieve a significant accuracy in duplicate bug report detection.

To improve software quality, developers allow the users of the system to submit bug reports that can be generated by using existing tools such as Bugzilla. Users can mention various things like, a description of the bug, the components that are affected by the bug and the extent of severity of the bug. Based on such factors priority is assigned to the bugs. Priorities are assigned because the resources are limited so the bug reports are managed based on the priority. However, currently, the whole procedure is a manual one. This motivated us to investigate other automated approaches, that would recommend a priority level based on information available in the bug reports. The information can be based on multiple factors like textual, author of the report, other similar bug reports already submitted, severity, product, or any other factors which can drastically affect the priority of the report. These factors are extracted as features which are then used to train a discriminative model via a classification algorithm that handles ordinal class labels. Experiments on more than thousands bug reports from Eclipse show that such approaches can outperform baseline approaches in terms of average F-measure by a relative improvement of up to 209%. [7].

We went over many research studies that have been carried out on detecting specially the configuration bugs. We studied how in one of the paper the line-of-work was extended that identifies only the configuration bugs as the specification can help developers reduce the debugging efforts and focus their effort on checking configuration files rather than checking the source code. The related work in this direction is by Arshad at al. [13] and Xia et al.[14]. Xia et al. use Arshad et al.'s method as a baseline and the results show that the proposed approach improves FL-score of Arshad et. Al.'s method.

3. HYPOTHESIS

Duplicate bug report could potentially provide different perspectives to the same defect enabling developers to better fix the defect in a faster amount of time. Still, there is a need to detect bug reports that are duplicates of one another. Use of recent advances in information retrieval community can be made to retrieve similar documents from a collection [4]. A model that contrasts duplicate bug reports from the non-duplicate bug reports is built to extract similar bug reports, given a query bug report under consideration.

4. RELATED WORK

Most of the techniques described in every paper that we studied have somewhere used the Information Retrieval approach due to the prevalence of natural language artifacts. Binkley et al. [5] applied a variety of IR techniques, which includes latent semantic indexing (LSI) which is a generative probabilistic model for sets of discrete data proposed by a mathematical theory of data analysis (FCA) using formal contexts and concept lattices on different software repositories. Many of the software problems have been addressed like fault prediction, developer identification for a task, assisting engineers in understanding unfamiliar code, estimating the effort required to change a software system and refactoring. Marcus et al. have used LSI to map the concepts expressed by the programmers to the relevant parts of the source code [6]. Their method is built upon finding the semantic similarities between the queries and modules of the software. Another generative model is approached for documents in which each document is related to a group of topics. A convexity-based variational approach for interference is demonstrated that is a fast algorithm with reasonable performance.

A labeled topic extraction method based on labeling the extracted topics from commit log repositories using non-functional requirement concepts is implemented. This method is based on LDA topic extraction technique and non-functional requirements concept is selected.

We read and summarized another related work: Bug report Deduplication. Just et al. [18] proposed that the current bug tracking systems have defects which cause the IR processes to be less precise. The author concluded that issue trackers should improve their interfaces to augment the information they already provide to the developers. Nagwani et al. [19] provide two different definitions of similar and duplicate bugs. Two bugs are similar when the same implementation behavior is required for resolving two bugs. Two bugs are duplicate when the same bug is reported by using different sentences in the description. Some similarity measures are preset if all the similarity measures meet the thresholds the bugs are duplicates. The bug reports are similar if only some of the measures are met.

In a method, proposed by Jalbert et al [20] bug reports are filtered based on an automatic approach. He introduced classifier for incoming bug reports which combine the categorical features of the reports, textual similarity metrics, and graph clustering algorithms to identify duplicates. Another approach was that of Wang et al. [21] who used natural language information accompanied by execution information to detect duplicate bugs, evaluated on the Firefox and Eclipse bug repositories. Reports are divided into three groups: run-time errors, feature requests, and patch errors. This approach shows some promise behind using contextual information as they achieve better performance than relying solely on natural language information.

For the paper we read on a more accurate retrieval of duplicate bug reports by Sun , Lo, Khoo and Jiang, Sun et al. [2] proposed a new text-similarity based duplicate bug-report retrieval model based on BM25F[3] which is a document similarity measurement method built upon to- idf. Other categorical information from the bug reports like a product, priority, and type are utilized to retrieve duplicate bug reports, in addition to textual features of bug reports. The evaluation of their method to see if the correct duplicate is a candidate or not is based on the list which consists of candidate duplicate bug reports for every bug report marked as "duplicate" by the triager. The triager detects if a bug report is a duplicate; if it is, the triager marks this report as a duplicate report and the first report as the master report.

The third related work we chose is on DupFinder, which consists of a client-side component and a server-side component. The client-side handles interaction with the user and communicates with the server-side component to retrieve relevant bug reports given a user query, which is a combination of the texts that appear in the summary and the description fields of a new bug report.
Bug report entry page is the page where a user can enter the details of a bug that the user intends to report. The client-side component is added to the bug report entry page. The server-side component compares the user query with the existing bug reports and outputs a list of most similar reports. Algorithm 1 shows the pseudocode of an algorithm for finding duplicate bug reports, which is implemented in the server-side component. It accepts as input a new bug report NewBr, k most recently created bug reports RecentReports, and a number of most similar bug reports to return n. The algorithm returns a ranked list of n bug re- ports in RecentReports that are most similar to NewBr. In lines 1-4, it concatenates the text in the summary and description fields of NewBr, performs text preprocessing on the concatenated text, and creates a vector space model (VSM) representation (which is a vector of weights) from the preprocessed text.

Algorithm 1 FindDuplicate

```
Input: NewBr: a new bug report
       RecentReports: k most recent bug reports
       n: number of most similar bug reports to return
Output: top-n most similar bug reports
1  NewBrTokens ← Tokenize(NewBr.Summary, NewBr.Desc)
2  Remove stop words from NewBrTokens
3  Stem each word in NewBrTokens
4  NewBrVSM ← ConstructVSM(NewBrTokens)
5  foreach instance Br ∈ RecentReports do
6      BrTokens ← Tokenize(Br.Summary, Br.Desc)
7      Remove stop words from BrTokens
8      Stem each word in BrTokens
9      BrVSM ← ConstructVSM(BrTokens)
10     Br.Sim ← CosineSimilarity(NewBrVSM, BrVSM)
11 end
12 Sort Br in RecentReports by Br.Sim
13 return top-n most similar bug reports
```

In lines 5-9, it also performs text concatenation, text preprocessing and constructs a VSM representation from the summary and description fields of each report in RecentReports. In line 10, it computes the cosine similarity between the VSM representation of the new bug report and the VSM representation of each bug report in RecentReports. In line 12, it then sorts bug reports in RecentReports based on their cosine similarity scores. Finally, in line 13, top-n bug reports with the highest scores are returned. By default, we set k and n to 100 and 5, respectively. Users might set k to a larger number to reduce the risk of not identifying duplicates of older bugs.

The client-side of DupFinder is implemented by overriding a template of Bugzilla that renders the user interface that allows users to input new bug reports. The server-side of DupFinder is implemented as a new web-service that is called by the client-side. These follow the standard procedure specified by Bugzilla to implement a new extension.

5. CHECKLISTS

The following can be the checklists that can be used to design the analysis of a duplicate bug report.

1. Summary: Summarized description of a bug. Typically this summary only contains a few keywords.
2. Description: Long description of a bug. Typically this would include information that would help in the debugging process including the reported error message, the steps to reproduce the error, etc.
3. Product: The product which is affected by the bug.
4. Component: The component which is affected by the bug.
5. Author: The author of the bug report.
6. Severity: The estimated impact of a bug as perceived by the reporter of the bug. There are several severity labels including blocker, critical, major, normal, minor, and trivial. Aside from these severity levels, there is one additional severity level that denotes feature requests, i.e., enhancement. In this study, we ignore bug reports with this severity label as we focus on defects and not feature requests.
7. Priority: The priority of a bug to be fixed which is assigned by a bug triager. When the bug report is submitted, this field would be blank. The triager would then decide an appropriate priority level for a bug report. There are five priority levels: P1, P2, P3, P4, and P5.

6. DATA

The work by Sun et al. has shown that REP, an advanced IR approach, outperformed state-of-the-art ML approaches in term of both accuracy and time efficiency. It is customized from BM25F to take into account the long bug reports and the meta-data such as the reported product, component, and version. The key assumption in REP is based on a high textual similarity between duplicate bug reports. However, in practice, it is popular that the bug reports can be filed by multiple reporters who could describe the same technical issue(s) in different phenomena via different terms. With different input data, usage environments or scenarios, an erroneous behavior might be exposed as different phenomena (e.g. different outputs, traces, or screen views). Moreover, different reporters might use different terminologies and styles, or write about different phenomena to describe the same issue(s). Thus, duplicate bug reports might not be very textually similar. In those cases, REP does not detect them well. The following is the data a typical bug report consists of.

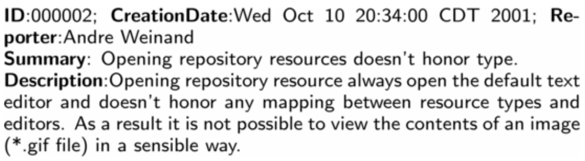
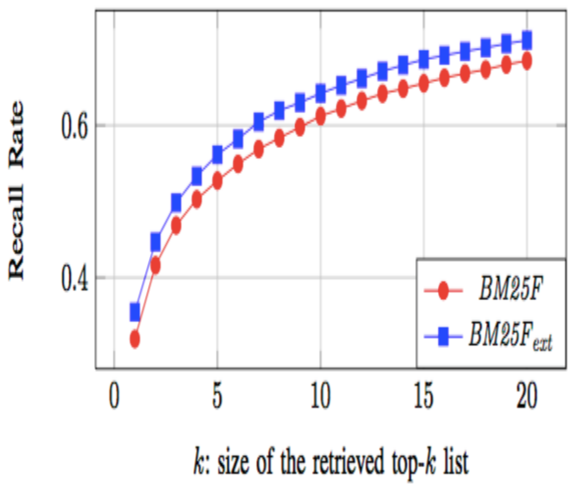

ID:000002; **CreationDate**:Wed Oct 10 20:34:00 CDT 2001; **Reporter**:Andre Weinand
Summary: Opening repository resources doesn't honor type.
Description:Opening repository resource always open the default text editor and doesn't honor any mapping between resource types and editors. As a result it is not possible to view the contents of an image (*.gif file) in a sensible way.

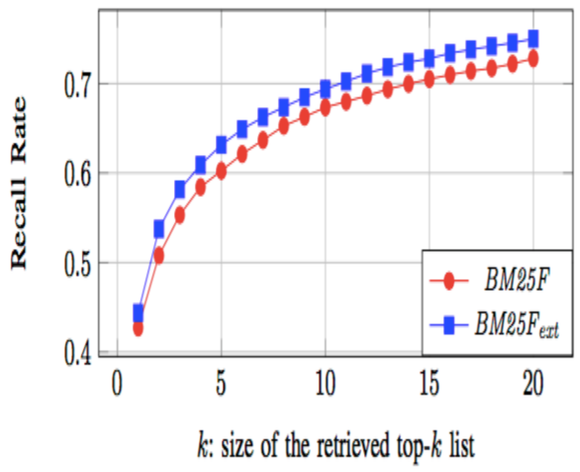
Figure 1: Bug Report BR2 in Eclipse Project

7. BASELINE RESULTS

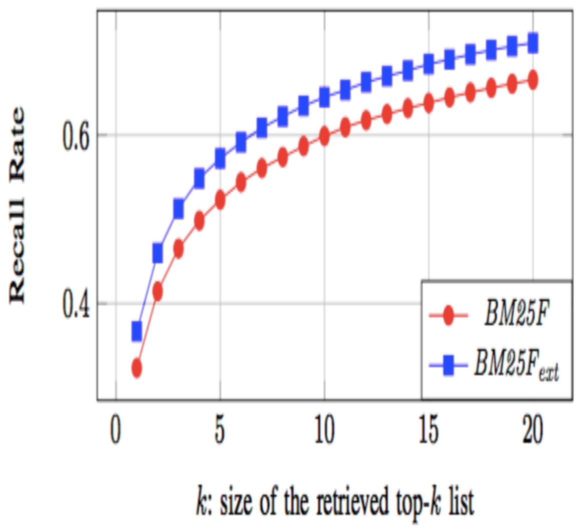
In paper 2 a prototype is built to validate the effectiveness of the extension to BM25F and their new retrieval function, and have applied it to the bug repositories of three large open source projects, OpenOffice, Mozilla, and Eclipse. MAP is a single-figure measure of ranked retrieval results independent of the size of the top list. It is designed for generally ranked retrieval problem, where a query can have multiple relevant documents. However, duplicate bug report retrieval is special as each query (duplicate report) has only one relevant document (master report). The case studies serve two purposes: the first is to validate the effectiveness of BM25F(ext) over BM25F; the second is to compare the retrieval performance of the proposed retrieval function REP, previous retrieval model based on SVM [11], and the work by Sureka and Jalote in [12].



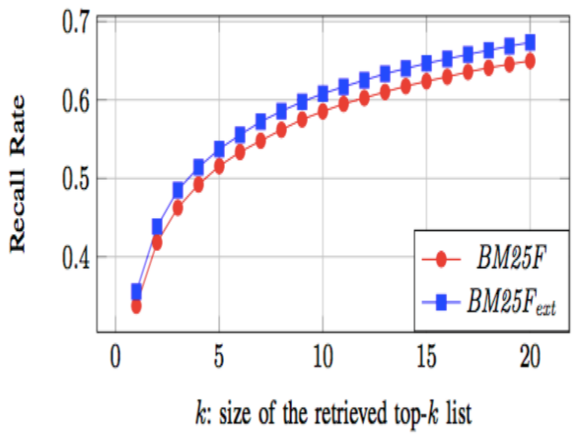
(a) OpenOffice



(c) Eclipse



(b) Mozilla



(d) Large Eclipse

8. STUDY INSTRUMENTS

In paper 2 that we studied the authors have evaluated their approach on bug report datasets of Mozilla which include subprojects such as Firefox (web browser), Thunderbird (email client), and Eclipse (Integrated development environment) and Open Office. The effectiveness of the BM25F extension is compared against a standard BM25F method. On each dataset, the proposed BM25F-extension performs constantly better than BM25F and it gains 4%–11%, 7%–13%, 3%–6% and 3%–5% relative improvement over BM25F on Open Office, Mozilla, Eclipse and Large Eclipse datasets respectively. In paper 3, Support Vector Machine (SVM) is an approach to building a discriminative model based on a set of labeled vectors. Based on positive class and negative class panes, SVM tries to build a hyperplane which finds the difference between the two planes with least margins. The paper used the libsvm library to do so. In the fifth paper, we studied issue-tracking system. Many software projects provide services for users to report bugs, and to store these reports in an issue-tracking system. Bug-tracking systems, such as Bugzilla and Google's issue tracker, enable users and testers to report their findings in a central repository with a short description and a longer summary, akin to a message subject line and body, as well as tool-specific additional information. The system uses this information to categorize and, possibly, further annotate the bug report. This enables developers to query for bug reports based on a combination of textual and categorical (attribute-based) queries.

In paper 8, we came across a new algorithm,EFS predictor. This algorithm applies ensemble feature selection on the natural-language description of a bug report. It uses different feature selection approaches (e.g., ChiSquare, GainRatio and Relief) which output different ranked lists of textual features. Then, to obtain a set of representative textual features, EFSPredictor assigns different scores to the features given by these feature selection approaches. Next, for each feature, EFSPredictor sums up the scores outputted by the multiple ranked lists and outputs the top features as the selected features. After which EFSPredictor builds a prediction model based on the selected features.

9. COMMENTARY

This section has our analysis of all the eight papers we studied.

Among all the papers, in paper 7 an automated approach can be considered as one of the best approaches which are based on machine learning that would recommend a priority level based on information available in bug reports. This approach considers multiple factors, temporal, textual, author, related-report, severity, and product, that potentially affect the priority level of a bug report. These factors are extracted as features which are then used to train a discriminative model via a new classification algorithm that handles ordinal class labels and imbalanced data. Experiments on more than a hundred thousand bug reports from Eclipse show that this approach can outperform baseline approaches in terms of average F-measure by a relative improvement of up to 209 %.

Also in paper 6, we studied how a tool DupFinder is developed, which is implemented as a Bugzilla extension, to search for duplicate bug reports[6]. Their goal was not to design a new algorithm but rather to implement an existing technique into a tool integrated into a bug tracking system that can be used by practitioners to help them deal with duplicate bug report problem. The tool is based on the unsupervised technique proposed by Runeson et al. [9]. The paper 5 that we studied posits a new evaluation methodology for bug- report deduplication [5], that improves the methodology of Sun et al.'s [3] by considering true-negative duplicate cases as well.

The first paper studied by us, introduced DBTM, a duplicate bug report detection model that takes advantage of not only IR-based features but also topic-based features [1]. In this paper, the authors extended Latent Dirichlet Allocation (LDA) [11] to represent the topic structure for a bug report as well as the duplication relations among them.

Paper 2 proposed a new duplicate bug report retrieval model [2] by extending BM25F introduced in paper 3 [3]. As per the paper, they engineered an extension of BM25F to handle longer queries. Paper 4 evaluated the performance of infoZilla on ECLIPSE bug reports. It correctly identified the presence of enumerations, patches, stack traces, and source code in bug reports.

10. NEW RESULTS

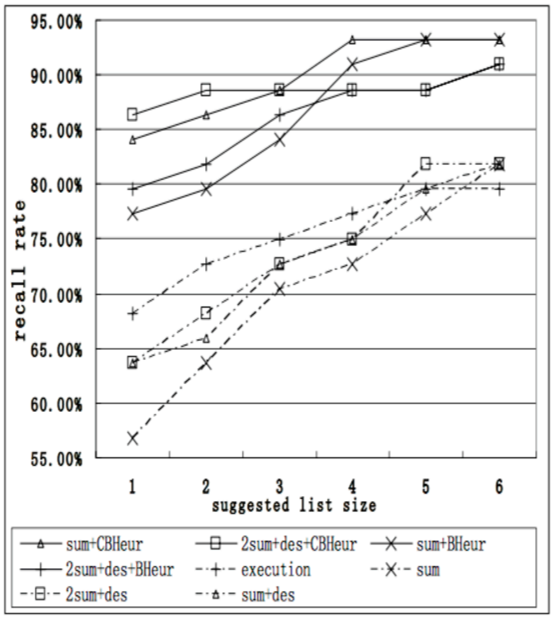
Paper 3[3] first considered a new approach for detecting duplicate bug reports by building a discriminative model that answers the question "Are two bug reports duplicates of each other?". This approach is considered as a base in almost every paper we studied. The model would report a score on the probability of A and B being duplicates. This score is then used to retrieve similar bug reports from a bug report repository for user inspection. The utility of the approach was investigated on 3 sizable bug repositories from 3 large open-source applications including OpenOffice, Firefox, and Eclipse. The experiment shows that the approach outperforms existing state-of-the-art techniques by a relative improvement of 17–31%, 22–26%, and 35–43% on OpenOffice, Firefox, and Eclipse dataset respectively. The issue in this approach was the accuracy was not achieved as targeted.

Paper 2 [2] proposed a new retrieval function fully utilizing not only text but also other information available in reports such as *product*, *component*, *priority*. A two-round gradient descent contrasting similar pairs of reports against dissimilar ones is adopted to optimize REP based on a training set and achieve a better accuracy. The utility of the technique was investigated on 4 sizable bug datasets extracted from 3 large open-source projects, OpenOffice, Firefox and Eclipse; and find that both and are indeed able to improve the retrieval performance. Particularly, the experiments on the four datasets show that BM25F(ext) improves *recall rate@k* by 3–13% and MAP by 4–11% over BM25F. For retrieval performance of REP, compared to previous work in paper 3[3] based on SVM, it increases *recall rate@k* by 10–27%, and MAP by 17–23%; compared to the work by Sureka and Jalote [12], performs with *recall rate@k* of 37–71% (1 ≤ k≤ 20), and MAP of 46%, which are much higher than the results reported in their paper.

11. PATTERNS

We studied a novel approach to assist triagers in detect- ing duplicate bug reports[14]. Unlike existing approaches, this approach further considers execution information. Furthermore, the approach employs two heuristics to combine the two kinds of information. The authors also calibrated and evaluated the approach on bug reports from the Eclipse and Firefox repositories. The experimental results show that, compared with the best performance of approaches using only natural language information, our calibrated approach (with the classified-based heuristic and using only the summary) leads to an in- crease of 11-20 percentage points and an increase of 18-26 percentage points in recall rates on the two experimental bug-report sets respectively.

The authors had four different combinations of parameters for their approach. As a comparison, they also considered three approaches using only the natural language information (i.e. summary only, equally-weighted summary and description, and double-weighted summary and description) and one approach using only the execution information. As before, used each of the 44 duplicate bug reports as the new bug report and the other 219 bug reports as the existing bug reports in evaluating each approach. The following Figure shows the result of the evaluation on Eclipse for suggested-list sizes of 1-6.



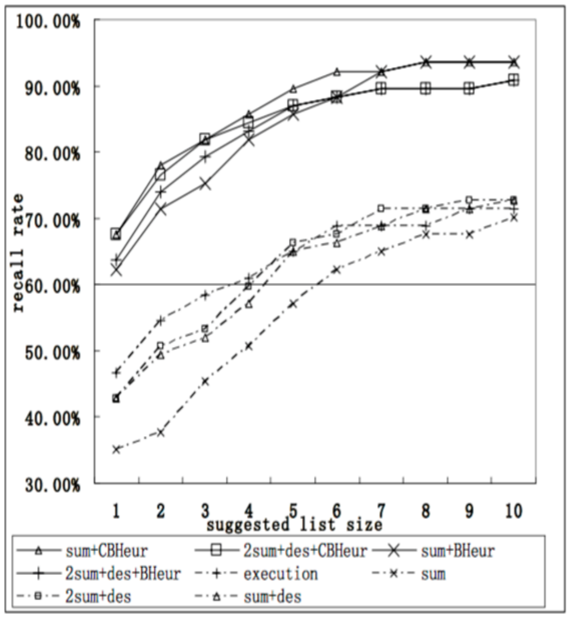
From above figure, following three observations are made. First, the four parameter combinations for the approach always outperform the other four approaches. As the classification- based heuristic always outperforms the basic heuristic, focus on the two combinations using the classification-based heuristic. When comparing the performance of the approach using both the summary and the execution information with the classification-based heuristic to the best performance of using only the natural language information, there is an increase of 11-20 percentage points for suggested-list sizes of 1-6. The increase of the other combination over the best performance of using only the natural language information is 7-22 percentage points.

Second, when the suggested-list size is small, using only execution information appears to outperform using only natural language information. When the suggested-list size becomes larger, using only natural language information becomes superior to using only execution information. We suspect the reason to be that execution information is more precise than natural language information. The precision in information leads to a better order of the retrieved bug reports. However, as bug reports with quite similar or even identical execution information are not guaranteed to be duplicate bug reports, using only execution information may become more likely to retrieve irrelevant bug reports than using only natural language information when suggested lists are large.

Finally, among the three approaches using only natural language information, the one using the double-weighted summary with the detailed description appears to achieve the best performance, and the approach that uses only the summary appears to perform the worst.

This observation is different from the situation of using natural language information together with execution information, where using only the summary becomes superior when the suggested-list size becomes larger. We suspect the reason to be that, without the other kind of information sources, only the summary cannot provide enough clues to achieve a high-quality set of retrieved bug reports or a high-quality ordering of the retrieved bug reports. As using the double-weighted summary with the detailed description performs slightly better than using the equal-weighted summary and description, this observation once again confirms the finding of Runeson et al. [9].

The authors designated the earliest bug report in a group of duplicate bug reports as the target report. They marked the duplicate bug reports whose target report was not in the experimental bug-report set as “unique” instead of “duplicate”. Under this strategy, there were totally 77 duplicate bug-report pairs in our evaluation.



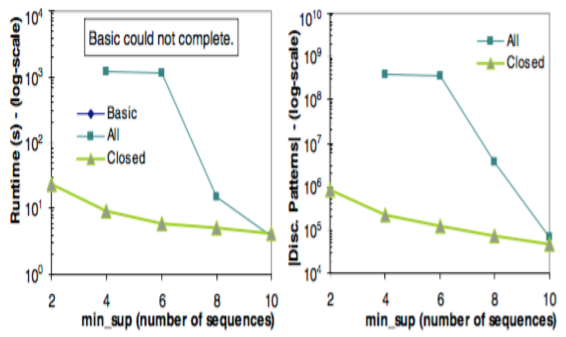
The results of the evaluation on Firefox are shown in the above Figure. We make similar observations from this figure as observed in Figure for Eclipse. The results can confirm almost all the findings in the evaluation on Eclipse. First, the four combinations for approach outperform the other four approaches. Unlike the results on Eclipse, among the four combinations of the approach, the one using the summary and the execution information with the classification-based heuristic outperforms the other combinations. This combination achieves recall rates of 67%-93% for suggested-list sizes of 1-10. However, there is a similar trend in both Figures: when the suggested- list size is small, our approach that uses only the summary has no advantage over the approach that uses the double-weighted summary with the detailed description. The advantage becomes significant only when the suggested-list size becomes larger.

Paper [10] used mined patterns as features to detect whether two bug reports are duplicate of each other or not and implemented a case study.

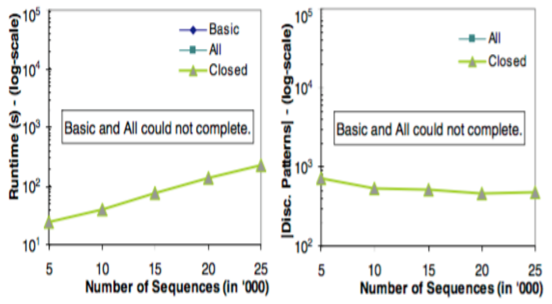
In this work, they proposed a new problem of mining from a database of labeled sequence pairs. The major interest in mining dyadic sequential patterns that appear frequently in the database, and could discriminate between two classes, +ve and -ve. Rather than mining all possible patterns satisfying the above criteria, the authors only mine a compact representation of such patterns referred to as closed patterns.

To realize the solution, we propose a new search space traversal strategy, canonical representation of patterns, and a pruning strategy based on the canonical representation of patterns, which avoids the generation and test of many redundant patterns. The authors also propose a new projected database data structure to address the issue with having two sequences in a pattern causing two possible ways to project a pattern on a sequence pair. They propose a number of new search space pruning strategies that leverage anti-monotonicity properties on support and the upper bound of discriminative score. They also present a closure checking property that could detect closed patterns on the fly and a new property to eliminate non-closed patterns enmasse. They embed the above strategy, data structure, and properties in several algorithm variants.

Experiments were performed on synthetic and real datasets to test the scalability and utility of our mining solution. It showed that mining closed patterns could be much faster than mining all frequent and discriminative patterns. The latter could also be much faster than our baseline solution. On many settings, the closed variant can complete, while all and baseline cannot. Patterns mined from the real software bug report dataset show that mined patterns could be used to detect duplicate bug reports.



Varying the minimum support threshold $min\ sup$ on the real dataset: runtime (left), and number of patterns (right)



Varying the number of sequences D with $min\ sup$ = 60, $PNum$ = 30, and $PSize$ = 30: runtime (left), and number of patterns (right)

The same dataset were used to mine patterns shown in Figure. The authors split the dataset into training and test. Given a training data containing pairs of bug reports, they extract features and learn a discriminative model.
They used LIBSVM[22] with probability estimates as the classification model. Classification accuracy, defined as the percentage of test cases correctly classified, is used as one mea- sure. The measure AUC which is the area under a ROC curve is also used. ROC curve shows the trade-off between true positive rate and false positive rate for a given classifier. A good classifier would produce a ROC curve as close to the top-left corner as possible. AUC is a measure of the model accuracy, in the range of [0,1.0]. The best possible classifier would generate an AUC value of 1.0.

The three feature sets for LIBSVM classification:

1. Single Tokens. We use a vector corresponding to the

set of tokens appearing in each pair of bug report.

2. Dyadic Patterns. we use a vector corresponding to the appearance/non-appearance of mined patterns in each pair of bug report. Patterns are mined from the training set with the minimum support and discriminative

score thresholds set at 2 and 0.0001 respectively.

3. Both. Combination of the above two feature sets.

The results for the three feature sets in terms of accuracy and AUC are shown in Table 4.
The result shows that by using dyadic patterns as features we are able to classify pairs of bug reports accurately with more than 80% accuracy and 0.90 AUC.
Table : Accuracy: Duplicate Bug Report Detection

Configuration	Accuracy	AUC
Single Tokens	60.38%	0.65
Dyadic Patterns	82.86%	0.90
Both	81.23%	0.89

12. CONCLUSION

In this study of eight papers related to Duplicate Bug Report Detection, we have exploited the domain knowledge and context of software development to find duplicate bug reports. By improving bug deduplication performance companies can save money and effort spent on bug triage and duplicate bug finding. Paper 5[5] uses contextual word lists to address the ambiguity of synonymous software-related words within bug reports written by users, who have different vocabularies. It replicated Paper 2 Sun *et al.*'s [2] method of textual and categorical comparison and extended it by adding contextual similarity measurement approach. We found that by the inclusion of the overlap of context as features the contextual approach improves the accuracy of bug-report deduplication by 11.55% over Paper 2 Sun *et al.*'s [2] method.

We also concluded that various tools and frameworks can be integrated to predict the duplicate bug reports. In paper 7[7], a framework named DRONE is introduced to predict the priority levels of bug reports in Bugzilla. It considered multiple factors including temporal, textual, author, related-report, severity and product. These features are then fed to a classification engine named GRAY built by combining linear regression with a thresholding approach to address the issue with imbalanced data and to assign priority labels to bug reports. The result on a dataset consisting of more than 100,000 bug reports from Eclipse shows that our approach outperforms the baselines in terms of average F-measure by a relative improvement of up to 209 % (Scenario "No-P3").

Developers can use such tools as a recommender system to prioritize bugs to be fixed.

13. FUTURE SCOPE/ OUR RECOMMENDATION

For paper 2[2] In future, the authors plan to build an indexing structure of bug report repository to speed up the retrieval process. Along with that they also plan to integrate their technique into the Bugzilla reporting system. Also, the results are based on the bug report database of 4 projects only with more or less structured reporting. The techniques should be applied to another open-source as well as proprietary projects to check for its correctness and universal applicability.
Future work of the paper[5] is to implement the introduced method as an embedded tool in an issue-tracker to empirically investigate the role that this method can actually play in assisting the triagers and save their time and effort when looking for the duplicates of an incoming bug report. Future plans also include evaluating what makes a good context. This approach could also be applied to more modern, state-of-the-art bug deduplication techniques such as those by Nguyen et al. (2012) and then evaluated for any improvements. In paper 6[6], DupFinder only applies a single unsupervised learning algorithm proposed by Runeson et al. to find duplicate bug reports. Other unsupervised, as well as supervised learning approaches, should be incorporated for wider applicability. One of the major areas of enhancement in the future for DupFinder will be the implementation of supervised duplicate bug report detection techniques along with the current unsupervised techniques.
For paper 8 [8] Additional bug reports from more projects should be investigated to reduce threats to external validity. The design of additional solutions can help boost the effectiveness of EFSPredictor. The experiments have been conducted on 5 bug report datasets (i.e., accumulo, activemq, camel, flume, and wicket) containing a total of 3,203 bugs. The experiment results show that, on average across the 5 projects, EFSPredictor achieves an F1-score to 0.57, which improves the state-of-the-art approach proposed by Xia et al. by 14%. Although this is a significant result, the datasets on which the model has been tested is not sufficiently big and can be improved upon by testing on new datasets such as Amazon EC2 APIs.

14. REFERENCES

[1] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, David Lo, Chengnian Sun. 2012. Duplicate Bug Report Detection with a Combination of Information Retrieval and Topic Modeling. Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference.

[2] Chengnian Sun , David Lo , Siau-Cheng Khoo , Jing Jiang, Towards more accurate retrieval of duplicate bug reports. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering.

[3] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, Siau-Cheng Khoo, A Discriminative Model Approach for Accurate Duplicate Bug Report Retrieval. In Proceedings of ICSE '10 Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1.

[4] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In MSR ’08: Proceedings of the 2008 international working conference on Mining software repositories, pages 27–30, 2008.

[5] Abram Hindle, Anahita Alipour, Eleni Stroulia, A contextual approach towards more accurate duplicate bug report detection and ranking. Proceeding MSR '13 Proceedings of the 10th Working Conference on Mining Software Repositories.

[6] Ferdian Thung, Pavneet Singh Kochhar, and David Lo, DupFinder: Integrated Tool Support for Duplicate Bug Report Detection. ASE '14 Proceedings of the 29th ACM/IEEE international conference on Automated software engineering.
[7] Published in Empirical Software Engineering, 2015 October, Volume 20 Issue 5, 1354-1383. Automated prediction of bug report priority using multi-factor analysis Yuan Tian, David Lo, Xin Xia and Chengnian Sun.

[8] Bowen Xu, David Lo, Xin Xia, Ashish Sureka and Shanping Li, EFSPredictor: Predicting Configuration Bugs With Ensemble Feature Selection. Conference: 2015 Asia-Pacific Software Engineering Conference (APSEC), Year: 2015.

[9] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In ICSE, 2007.

[10] D. Lo, H. Cheng, and Lucia. Mining closed discriminative dyadic sequential patterns. In EDBT, 2011.

[11] D. M. Blei, A. Y. Ng, M. I. Jordan. Latent Dirichlet Allocation. J. Mach. Learn. Res., 3: 993-1022, 2003.

[12] A. Sureka and P. Jalote, “Detecting duplicate bug report using character n-gram-based features,” in *Proceedings of the 2010 Asia-Pacific Software Engineering Conference*, 2010, pp. 366–374.

[13] F. Arshad, R. J. Krause and S. Bagchi

"Characterizing configuration problems in java ee application servers: An empirical study with glassfish and JBoss"
Software Reliability Engineering (ISSRE) 2013 IEEE 24th International Symposium, pp. 198-207.
[14] X. Xia, D. Lo, W. Qiu, X. Wang and B. Zhou
"Automated configuration bug report prediction using text mining"
Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual, pp. 107-116,

[15] R. Nallapati. Discriminative models for information retrieval. In SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval, 2004.

[16] D. Binkley and D. Lawrie, "Information retrieval applications in software maintenance and evolution," *Encyclopedia of Software Engineering*, 2009.

[17] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. IEEE, 2004, pp. 214–223.

[18] S. Just, R. Premraj, and T. Zimmermann, "Towards the next generation of bug tracking systems," in *Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on*. IEEE, 2008, pp. 82–85.

[19]N.NagwaniandP.Singh,"Weight similarity measurement model based, object oriented approach for bug databases mining to detect similar and duplicate bugs," in *Proceedings of the International Conference on Advances in Computing, Communication and Control*. ACM, 2009, pp. 202–207.

[20] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE, 2008, pp. 52–61.

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