Detecting Duplicate Bug Reports with Software Engineering Domain Knowledge*

Karan Aggarwal, Finbarr Timbers, Tanner Rutgers, Abram Hindle * †, Eleni Stroulia, and Russel Greiner

Department of Computing Science, University of Alberta, Edmonton, Canada

SUMMARY

Bug deduplication, i.e., recognizing bug reports that refer to the same problem, is a challenging task in the software-engineering life-cycle. Researchers have proposed several methods primarily relying on information-retrieval techniques. Our work motivated by the intuition that domain knowledge can provide the relevant context to enhance effectiveness, attempts to improve the use of IR by augmenting with software-engineering knowledge. In our previous work, we proposed the *software-literature-context method* for using software-engineering literature as a source of contextual information to detect duplicates. If bug reports relate to similar subjects, they have a better chance of being duplicates. Our method, being largely automated, has a potential to substantially decrease the level of manual effort involved in conventional techniques with a minor trade-off in accuracy.
In this study, we extend our work by demonstrating that domain-specific features can be applied across projects than project-specific features demonstrated previously while still maintaining performance. We also introduce a hierarchy-of-context to capture the software-engineering knowledge in the realms of contextual space to produce performance gains. We also highlight the importance of domain-specific contextual features through cross-domain contexts: adding context improved accuracy; Kappa scores improved by at least 3.8% to 10.8% per project. Copyright © 2016 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: deduplication; duplicate bug reports; information retrieval; software engineering textbooks; machine learning; software literature; documentation.

1. INTRODUCTION

Modern software projects use issue-tracking systems to record bug/issue reports, a colloquial term for the issues that developers, testers, and users encounter while using a particular software system. Primarily, these tracking systems serve as a store of bug reports, stack traces, and feature requests, and are sometimes used to measure the developers' productivity based on their progress in addressing issues. Bug reports are usually written in natural language; as a result, the same issue can be described in different ways by the project developers and testers and the system users who encounter the issue. Typically the vocabulary used by developers differs from that used by users, and can vary among users depending on their level of technical sophistication. Currently, many projects are forced to use a triager, often an experienced developer, to "translate" bug reports into a more technical language, relevant to developers. Duplicate bug reports waste the triager's and developers

*All of the word lists and bug datasets used in this paper can be found online at: https://bitbucket.org/kaggarwal32/bug-deduping-dataset

*Correspondence to: Department of Computing Science, University of Alberta, Edmonton, Canada.
†E-mail: abram.hindle@ualberta.ca

time. If manual triaging effort could be reduced, developer productivity would be increased, as they would not have to consider multiple reports for the same bug and they would have more information about each bug report, enabling them to fix each bug faster.

Considerable research has been done on automated methods for detecting duplicate bugs. Prior works from Runeson *et al.* [21] and Sun *et al.* [23, 22] on *bug-report deduplication* —namely, the detection of duplicate bug reports— measure bug-report similarity considering the bug-report textual descriptions, as well as categorical bug attributes such as "component", "type", and "priority". These approaches typically use off-the-shelf document-similarity measures, applying them to bug reports. While effective, such approaches ignore an important aspect of the deduplication problem: bug reports are not New York Times articles —the traditional domain in the context of which IR methods have been formulated— but rather technical reports about software projects.

Alipour *et al.* [3, 2] exploited the technical nature of the bug reports by using software-engineering and project-specific context to improve bug-report deduplication efforts. By exploiting contextual features through comparing a bug report to terms referring to non-functional requirements or architectural descriptions, Alipour *et al.* improved on the bug-deduplication performance of Sun *et al.* [22]. This contextual method relied on manually constructed contextual word lists, and topics generated through supervised labelled Latent Dirichlet Allocation (labelled-LDA) on the project's bug descriptions. Labelled-LDA worked well but is an extremely effort-intensive process [10]. Alipour concluded that contextual features tend to reveal the relationship between the bug report text and concepts such as non-functional requirements and/or architectural modules. These relationships can be exploited in bug-report deduplication when different terminology is used to describe the same scenario; not only could one compare the text between bug reports, but also their corresponding non-functional and architectural contexts as well. Continuing on this work, and aiming to reduce the cost of constructing these contextual features, we introduced a method for generating these features from software-engineering literature, which proved cost effective, generalizable, and easy to use [1]. Our method performs at par with unsupervised LDA approaches and is marginally worse than the labelled LDA approach, with far less manual labour involved.

Continuing on this line of work, the study reported in this paper examines the relevance of different layers of abstraction in the contextual space, capturing multiple essential aspects of software development and processes. We include domain-specific contextual features, which are more abstract than the project-specific context previously used, and thus more general. This general context can be used across different projects, while still maintaining project/domain level specificity. We show the effectiveness of the domain knowledge and domain awareness by using out-of-context features and comparing the results against our previous work. Second, we use our hierarchy of contexts to demonstrate the utility of including additional layers of domain-specific features. Finally, we use the software body of knowledge guide (SWEBOK) as an additional source of contextual features from more general software-engineering literature to demonstrate the superiority of general contextual features given their ease-of-sharing across projects.

In this paper we address the following research questions:

- RQ1: Does project documentation help deduplicate bug reports?
- RQ2: Does domain-specific knowledge help deduplicate bug reports?
- RQ3: Does general software engineering knowledge help deduplicate bug reports?
- RQ4: Does using more than one context (project-specific, domain-specific, or general) improve deduplication performance?

We improve upon our previous work by enhancing the generality and reuse potential of our contextual technical-literature method through using the domain-specific contextual features. The method is easy to use, as developers can use already extracted features, or, should they need to generate their own features, the developers have only to label textbook and project documentation chapters to extract the features in question, instead of having to extract the features themselves. The labelled LDA method used by Alipour *et al.* [3, 2] required the features to be extracted manually. Our method is thus easily generalizable, and allows for contextual features to be easily shared. The

marginal cost of using already extracted features in the form of word lists, such as those provided by the authors, is merely the cost of downloading the data.

2. RELATED WORK

Most bug-deduplication methods use textual analysis to detect duplicate bug reports. Runeson *et al.* [21] used *natural language processing* (NLP) techniques to detect 66% of the duplicate reports of Sony Ericson Mobile Communications. Bettenburg *et al.* [5] used machine-learning classifiers — support-vector machines and naïve bayes classifiers— to triage the reports based on a word vector representation of the report titles and descriptions and obtained roughly 65% accuracy.

Jalbert *et al.* [12] used the categorical features of bug reports in conjunction with textual-similarity measures and graph-clustering techniques to filter out duplicate reports. Their method was tested on a dataset of $29,000$ bug reports from the Mozilla Firefox project and was able to filter out 8% of the duplicate reports. We test our method against the same dataset. Tian *et al.* [27] extended this work with extensive similarity measures to substantially improve accuracy.

Wang *et al.* [9] proposed a method using NLP techniques to extract data from execution traces on the Eclipse project's set of bug reports; their methodology relies on the execution traces being manually extracted, which is extremely time-intensive. We also test our method against the Eclipse dataset.

Surekha *et al.* [25] used an *n*-gram based textual model on the Eclipse dataset to report top-*k* bug reports that could potentially be duplicates of a given bug report. Sun *et al.* [22] built on this work to propose a new model centered around around the BM25F score, using the term frequency-inverse document frequency (TF-IDF) vector-space model with BM25F as a similarity score between reports. In addition, they used categorical features such as priority and severity to produce substantial improvements over previous methods. Sun *et al.* [22] sorted the reports into different "*buckets*" corresponding to the underlying bugs, and focused on sending incoming duplicate reports to the appropriate bucket. They evaluate their method by comparing the list of top-*k* potentially duplicate bug reports against the true-duplicates list for each bug, obtaining an improvement of 10-27% over Surekha *et al.*. One issue with this kind of evaluation is that only true-positives are queried. Bug reports with no duplicates are not considered by the evaluation methodology, and therefore true-negatives are not examined. This is a drawback since the ability to identify true-negatives is a desirable functionality of any deduplication process, in effect enabling developers to proceed with the bug fix confident that there is no other similar reports to consider. Nguyen *et al.* [19] added topic modeling to Sun's [22] work.

Alipour *et al.* [3, 2] improved upon the work of Sun *et al.* [22] by adding contextual features extracted using both labelled and unlabelled LDA generated word lists [10] to the method used by Sun *et al.*. Alipour *et al.* reformulated the task as detecting whether a given pair of bugs are duplicates or not. The use of LDA produced strong improvements in accuracy, increasing by 16% over the results obtained by Sun *et al.* [22]. Klein *et al.* [14] and Lazar *et al.* [15] have leveraged the same dataset against new textual metrics based on LDA's output to achieve an accuracy improvement of 3% over Alipour *et al.*. Their work is promising but relies on running LDA on the corpus itself, whereas some of the features described in this paper are extracted only once from textbooks and can be applied broadly to other software projects without any further extraction effort from the user. We apply the features here to four different projects without any further effort after the initial extraction.

Thung *et al.* [26] provide one of the very few implementations of the full end-to-end bug deduplication systems based on Runeson *et al.* [21]'s model.

3. METHODOLOGY

This section describes our contextual bug-deduplication method. First, we describe our processes for curating our datasets and contextual word lists. Next, we explain our process for extracting

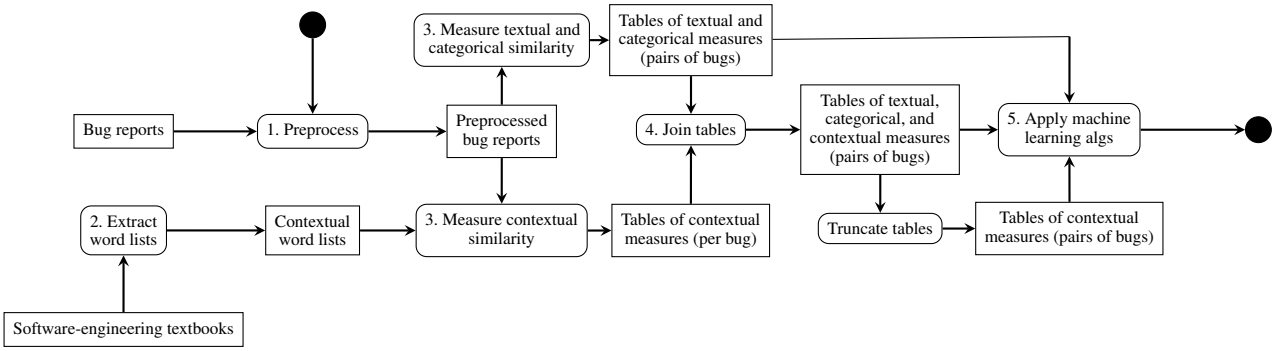

Figure 1. Workflow of the software-literature context evaluation methodology showing inputs and output. The sharp edged rectangles represent data and the rounded corner rectangles represent activities.

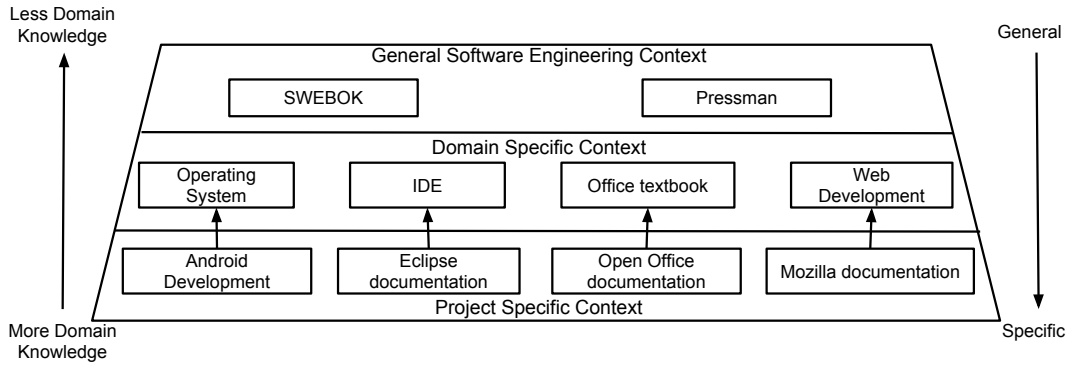

Figure 2. Hierarchy of the context with project-specific context at the bottom, domain-specific context in the middle layer, and general software engineering context at the top, in the increasing order of their generalizability from the bottom up. Arrows from the project-specific contexts to domain-specific contexts shows the relation between the project and domain-specific word lists.

Table I. Details of Datasets

Platform	#Bugs	#Duplicates	Duration	
			From	To
Android	37536	1361	2007-11	2012-09
Eclipse	43729	2834	2008-01	2008-12
Mozilla	71292	6049	2010-01	2010-12
OpenOffice	29455	2779	2008-01	2010-12

contextual features. Finally, we discuss our methodology for evaluating the effectiveness of these contextual features for accurate bug-report deduplication.

The Android, Eclipse, Mozilla, and Open Office bug report datasets used for analysis consist of $37,536$, $43,729$, $71,292$, and $29,455$ bug reports respectively. The Android bug reports are from Nov, 2007 to Sept, 2012; Eclipse for the year 2008; Mozilla for the year 2010; and Open Office for 2008 to 2010. Details are shown in Table I.

Each report contains the following fields: *Bug ID, description, title, status, component, priority, type, version, open date, close date,* and *Merge ID*. In the case of Mozilla, *dup_id* is used in place of *Merge ID*, as explained below. If a bug is a duplicate of another, the "status" field is marked as

"Duplicate," and the Bug ID of the duplicate report(s) are listed in the "Merge ID" field. This enables developers to determine how many bugs are duplicates, and reveal groups of duplicates bugs. Table II shows an example of two sample bugs containing information representative of a typical Android bug. Note that Bug `2282` is not a duplicate of any other bug(s), therefore its MergeID is empty, whereas Bug `14518` is a duplicate and has a non-empty MergeID assigned to it.

Figure 1 depicts the workflow of the *software-literature context method*. 1) The main starting point is to take bug-reports and pre-process them, but this assumes that contextual word lists have already been acquired or extracted. 2) Optionally one can extract new word lists from available literature. 3) Once the bug reports have been pre-processed, pairs of bug reports have their textual and categorical similarity measured, as well as their contextual similarity. 4) The similarities of contextual, categorical, and textual measures are joined together with the bug-report pairs. 5) Machine learning classification is applied to pairs of bugs leveraging the categorical, textual and contextual features.

3.1. Contextual-Features Extraction

Contextual features are organized in a hierarchy of contexts, as shown in Figure 2, according to their specialization/generalization relationships. The most general context, that of general software engineering, is at the top, followed by domain-specific context and project-specific context. These mid-level domain specific contexts can be viewed as software product-type contexts, that can be applied to software projects providing similar functionalities and features, and build on similar principles.

The following ten sets of word lists were used as contexts in this study, divided into three context levels. Across the ten sets, there are a total of 145 word lists. Each set of word lists took one graduate student approximately one half hour to create, for an approximate total of five hours spent creating word lists.

1. *General software-engineering context*: This context represents the general software-engineering practices and processes. Two literature sources have been used to extract two contexts.
   (a) *Pressman*: These word lists are extracted from Pressman's textbook [20]. The book was split into 13 different word lists corresponding roughly to the chapters of the book, e.g. architecture, UI design, formal methods, and testing.
   (b) *SWEBOK*: These word lists are extracted from the internationally accepted software body of knowledge guide [6] collaboratively developed by members of industry and academia. The book enumerates 15 areas of knowledge in software engineering such as requirements, design, maintenance, and testing. 15 word lists were extracted from these areas.
2. *Domain-specific context*: This context, though not directly related to any specific project, is much more meaningful to projects than the general software-engineering context. For example, for the Eclipse project, Java compilers/interpreters serve as an example that is not project specific, but rather, generic to the project's functionality. For each of the four projects we study, we include one such context.
   (a) *Operating system context*: These word lists were extracted from the chapters of Love [16] "Linux kernel development" to produce 13 word lists. The word lists describe features like devices, input/output, memory, and process management. The topics are all specific to the Linux operating system upon which Android is based.
   (b) *IDE context*: These word lists were extracted from a book on compilers for Java by Andrew *et al.* [4]. The documentation was split into 14 different word lists based on Java compiler specific concepts such as lexical scope, OOP, semantics, and garbage collection.
   (c) *Office context*: These word lists were extracted from the book that serves as a guide to using Microsoft Office [28]. The documentation was split into 11 different word

lists relating to various components, such as designing, work-flows, applications, and syncing.

(d) *Web context*: These word lists were extracted from Connolly *et al.* [8]'s book on web development. The documentation was split into 15 different lists relating to various components such as HTML, Javascript, CSS, and security.

3. *Project-specific context*: This is the lowest, most specific, level of context, and represents the project-specific contexts, directly related to the projects being studied. For each of the four projects, one such context has been considered.

   (a) *Android development*: Ten word lists for the Android project were extracted from Murphy [18] describing features such as widgets, activities, and databases. The lists were all related specifically to Android application development.

   (b) *Eclipse documentation*: The Eclipse platform documentation for Eclipse 3.1 [13] was split into 19 different word lists relating to Eclipse, such as debugging and IDE features.

   (c) *Open Office documentation*: The developer documentation for Open Office 3.0 [24] was split into 22 different word lists relating to various components such as spreadsheets, text documentation, database access, API design, and GUI.

   (d) *Mozilla documentation*: Unlike Eclipse and Open Office, there is no one central documentation for Mozilla products. The online developer guide for Mozilla [17] consists of several webpages, with very short descriptions catering to an online audience. The documentation was split into 13 different word lists relating to using various components such as browser, Javascript, debugging tools, and testing.

All these contexts and their word lists were extracted by labelling chapters on the basis of the software-engineering processes such as maintenance or testing. During labelling, text from similar chapters were grouped together under a single label. For instance, chapters titled "Software testing techniques", and "Software testing strategies", were grouped under *software testing*.

The office domain context is problematic but Open Office seeks to clone much of Microsoft Office's functionality thus due to the lack of a general text on office software. Thus we opted for the inspiration of Open Office's functionality.

This process of labelling chapters was done by a single person who is familiar with software development and software development processes. Since, this is aimed at minimizing time efforts as well as allowing developers to use the literature they think is relevant to the domain, the labelling process was done by a single person.

To build these lists, the frequency of word occurrences in the text under each label was recorded. Then, every word that appeared on a comprehensive list of stop words [7] was removed. Every remaining word that occurred less than 100 times was also removed. The threshold of 100 was used as it appeared to the authors to be the cut-off point between domain-specific language and generic words. This choice is relatively arbitrary and what a good threshold is will likely depend on the size of the documents and the breadth of their vocabulary. No frequency cutoff was used for the extraction of features from documentation, as documentation tends to be far more concise than other forms of technical literature. Hence, the three project specific lists—Eclipse, Mozilla, and Office documentation word-lists were extracted without any frequency cutoffs. It took about half of a person-hour to construct each of these five word lists. These word lists were used for generating contextual features as described in Section 3.5. The scripts to generate these word lists are included in the data-set data-dump.

3.2. Bug-Report Preprocessing

The bug reports were pre-processed following the methodology used by Alipour *et al.*. [3]

- Bugs that were missing significant amounts of information were discarded—bugs without Bug IDs as well as bugs marked as a duplicate where the corresponding duplicate Bug ID was not found in the repository—were removed.

Table II. Example Bug Report Information

BugID	Component	Priority	Type	Version	Status	MergeID
2282	Applications	Medium	Defect	1.5	Released	
14518	Tools	Critical	Defect	4	Duplicate	14156

Table III. Example Contextual features table

BugID	Process	Manage	Design	Test	...	Re-Eng
14518	0.377	6.887	2.847	4.997	...	0.753
14516	0.377	6.887	2.847	4.997	...	0.753
14690	0.681	7.923	3.175	7.954	...	1.718

Table IV. Example Textual and Categorical feature table

$BugID_1$	$BugID_2$	$BM25F_{uni}$	$BM25F_{bi}$	Product	Component	Priority	Type	Version	Class
14518	14516	1.484	0	0	1	1	1	1	dup
7186	7185	1.440	0.16	0	0	1	1	0	non

Table V. Example of final features table. Only selected features are shown here for representational purposes— $BM25F_{unigram}$ from textual similarity features, Version from categorical features, and Process/Re-Engineering from categorical features.

Bug pair		Features							
$BugID_1$	$BugID_2$	$BM25F_{unigram}$	Vers	$Proc_1$	$ReEng_1$	$Proc_2$	$ReEng_2$	Cosine_Similarity	Class
21756	21750	10.78	0	2.96	3.86	1.11	0	0.928	dup
8542	8541	3.07	0	0	1.20	0.56	1.80	0.926	non

- Stop words were removed from the description and title fields using a comprehensive list of English stop words. [7]
- The reports were organized into "buckets", as done by Sun *et al.* [22]. Each bucket contains a master bug report along with all duplicates of that report. The master bug report is the report with the earliest open time in that bucket.
- A bucket that contained a very large set of duplicate bugs was removed from the Android dataset. If the cluster was included, it would have strongly biased our results upwards, as such large clusters of duplicates are uncommon.

After the preprocessing step, three different subsets of the bug reports were constructed for each dataset, each containing a different ratio of duplicate to non-duplicate reports. The differing ratios are used to observe the effects of different ratios on accuracy. The same procedure was used in our previous work [1] as a robustness measure. The three subsets used include a set with 10%, 20% and 30% duplicates (as per Alipour *et al.*). In each case, random selection without replacement from the original dataset was used, selecting as many reports as possible while maintaining the desired ratios. There is no difference in relative performance among the datasets being compared, and there is no relative gain by using any particular ratio. Emphasis is placed on the 20% duplicate/ 80% non-duplicate split used by Alipour *et al.* in order to provide a comparison level of performance. We use the 20% to 80% ratio in order to provide easier comparisons with previously published results. We make no claim that the 80/20 is "better" or more realistic than the other ratios.

3.3. Textual Similarity Features

After the bug-report pairs are built, we compute the textual and categorical similarity of the bug pairs. While extracting textual/categorical features, reports were compared in a pairwise manner and similarity ratings were generated for each primitive field in each pair of reports. Each of the following comparison methods were adapted from the paper by Sun *et al.* [22] and were used in by Alipour *et al.* [3, 2].

Title and description fields were compared between bug reports using a customized version of BM25F, including both a unigram comparison (words treated individually) and bigram comparison (words treated in pairs).

3.4. Categorical Similarity Features

Categorical fields (component and type) were compared using a simple binary rating resulting in a value of 1 if matching and 0 otherwise. The comparison also included a comparison for a product field, however this field was not specified in the Android reports, and hence the product-field comparison was omitted on the Android reports. The two remaining fields (priority and version) were compared using a simple distance metric resulting in a value between 0 and 1 (where 1 indicates identical values). A total of *seven* textual and categorical features were obtained.

The exact formulas used are below, where d_1 and d_2 indicate sample bug reports [22, 3]:

$$\text{textual}_1(d_1, d_2) = BM25F(d_1, d_2)_{unigram} \tag{1}$$

$$\text{textual}_2(d_1, d_2) = BM25F(d_1, d_2)_{bigram} \tag{2}$$

$$\text{categorical}_1(d_1, d_2) = \begin{cases} 1 & \text{if } d_1.prod = d_2.prod \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

$$\text{categorical}_2(d_1, d_2) = \begin{cases} 1 & \text{if } d_1.comp = d_2.comp \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

$$\text{categorical}_3(d_1, d_2) = \begin{cases} 1 & \text{if } d_1.type = d_2.type \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

$$\text{categorical}_4(d_1, d_2) = \frac{1}{1 + |d_1.prio - d_2.prio|} \tag{6}$$

$$\text{categorical}_5(d_1, d_2) = \frac{1}{1 + |d_1.ver - d_2.ver|} \tag{7}$$

In the above equations, prod, comp, prio, type, and ver refer to the product, component, priority, type (defects or enhancement types), and version field in the bug reports, respectively. Equation (1) applies for unigram scores using BM25F while equation (2) uses bi-grams to calculate the similarity scores. An example of the textual and categorical features table can be seen in Table IV.

3.5. Contextual Features and Table Generation

After the computation of the textual and categorical features of the pairs of bug reports, contextual features were constructed using BM25F similarity scores of bug reports with word lists.

BM25F scores for word lists: The bug-report contextual features were computed using the BM25F algorithm for the comparison of the word lists with bug-report titles and descriptions. Each bug report was compared to the set of word lists generated for the given context (as mentioned in Section 3.1), where each word list was treated as text. This process results in a set of features corresponding to the word lists of that context. For example, the general software-engineering context contains 13 word lists, so there are 13 software-engineering contextual features for each bug report. This procedure was repeated for each of the contextual categories under investigation: general software engineering, domain-specific, and project specific. An example of the general software-engineering context features can be seen in Table III.

Feature table generation: Next, the contextual features for individual bug reports are calculated. Using these tables, a comparison feature table is constructed for pairs of bugs. Initially, pairwise

comparisons are generated for the textual and categorical features, as discussed in Section 3.3. The contextual features are subsequently added for each of the bug reports along with a cosine similarity feature based on the contextual feature vectors of the two bug reports. With n as the number of contextual features generated using n word lists for each bug report and Ci (with $i = 1, 2$) being the vector representation of the similarity metrics computed for each of the features, the cosine similarity of two reports is defined as:

$$\text{cosine\_sim}(b_1, b_2) = \frac{\sum_{i=1}^{n} C1_i \cdot C2_i}{\sqrt{\sum_{i=1}^{n} C1_i^2}\sqrt{\sum_{i=1}^{n} C2_i^2}} \tag{8}$$

The resulting table is an all-features table containing the textual, categorical, and contextual ratings for all pairs of bugs. Table V illustrates the final table representation. The labels $Proc_1$, $Proc_2$ stand for the context $Process$ in TableIII for the bug reports 1 and 2 being considered in this pair.

The textual features can be interpreted as the similarity between two bug reports based on purely their description, whereas contextual features are the similarity between a bug report and the context at hand, for example, UI design. If two bug reports have many words in common, but do not have common context, we expect our features to capture that commonality. In case the pair of bug reports has a lot of common words with the contextual word list but not with each other, it will be captured by the textual similarity score calculated by BM25F. Previous works [3, 2, 1] show that contextual features are able to capture such information over the conventional textual similarity methods [22], increasing accuracy by over 11-12% [3]. Hence, this approach captures both bug pair similarity as well as their contextual similarity.

Tables containing only contextual-feature ratings for all pairs of bugs were also generated to evaluate the effects of training on only contextual ratings. The tables were generated by simply removing the textual and categorical features from the all features table to see how these features perform on their own. Once the data was prepared, it was passed on to the machine-learning classifiers for training, testing, and evaluation.

3.6. Machine Learning and Evaluation Criteria

The data tables described in the previous section are meant to be provided as input to machine-learning algorithms, in order to produce classifiers that can recognize a pair of bugs as "duplicate" or "non-duplicate". Unlike in the work of Sun *et al.* [22], which queries only duplicate bugs, we consider true-negatives in the evaluation measure the accuracy of comparing two non-duplicates. This is especially important in scenarios where duplicates are not explicitly marked. The tables were provided as an input to Weka [11], which runs a number of standard machine-learning classifiers. The model obtained through Weka was tested to see how well it performed on the task of assigning the correct label to a pair of bugs — "duplicates," if the two bugs are duplicates or "non-duplicates," if they are not. 10-fold cross validation was used as an added robustness measure.

The performance of these models was evaluated in terms of accuracy and Cohen's Kappa coefficient. Accuracy is defined as the ratio of number of correctly classified instances to the total number of instances. Cohen's Kappa coefficient is a modified version of the accuracy score that attempts to compensate for blind luck. It measures the accuracy of a classifier in comparison to the probability of randomly assigning the correct classification. We follow the lead of Weka [11] and define Kappa in Equation (9), where $P(e)$ is the chance of correctly classifying the sample using the majority classification, i.e. using the ZeroR learner or a naïve Bayes classifier. The naïve Bayes classifier is one of the simplest classifiers, and using this definition for Kappa allows for a "classifier-to-classifier" comparison. In this specific case, $P(e)$ is equal to the number of non-duplicates expressed as a percentage of the data-set.

$$\kappa := \frac{\text{Accuracy} - P(e)}{1 - P(e)} \tag{9}$$

Table VI. Accuracy and Kappa scores using different contextual feature combinations for the Android dataset by using C4.5 decision tree algorithm.

In-context				
Project-specific context	Domain-specific context	General context	Accuracy	Kappa
Android development	Operating system	Pressman	**92.50**	**0.770**
Android development	IDE	Pressman	92.37	0.760
Android development	Office	Pressman	92.07	0.751
Android development	Web	Pressman	92.18	0.755
Android development	Operating system	SWEBOK	92.42	0.760
Android development	IDE	SWEBOK	92.30	0.759
Android development	Office	SWEBOK	92.07	0.750
Android development	Web	SWEBOK	92.39	0.760
Out-of-context				
Project-specific context	Domain-specific context	General context	Accuracy	Kappa
Eclipse documentation	IDE	Pressman	92.06	0.750
Mozilla documentation	Web	Pressman	91.96	0.745
OpenOffice documentation	Office	Pressman	91.82	0.739
Eclipse documentation	IDE	SWEBOK	92.00	0.747
Mozilla documentation	Web	SWEBOK	**92.16**	**0.754**
OpenOffice documentation	Office	SWEBOK	91.81	0.739
Best performing features from our previous work [1]				
Project-specific context	Domain-specific context	General context	Accuracy	Kappa
Labelled LDA	-	-	**93.62**	**0.799**
Android development	-	Pressman	92.36	0.756

C4.5 (Weka's J48) decision-tree classifiers were used with the default parameters to evaluate the performance of software-literature context method features. We chose C4.5 as it was the best performing classifier in our previous work [1], and thus allows for a direct comparison of results.

4. RESULTS

How well do our generated software-literature context features, extracted from documentation and textbooks, help answer the question, "are these two bug reports duplicates or not?" Using bugs from each project repository, tables with a ratio of pairs of duplicates to non-duplicates of 20-80 were sampled. The classification algorithms were applied on two different sets of features: the contextual features by themselves (software-literature context method), and the contextual features combined with the textual and categorical features (features from Sun *et al.*[22]). The results are summarized in Tables VI, VII, VIII, and IX.

Each dataset has three hierarchical contexts —general software engineering, domain-specific, and project-specific. In order to evaluate the effectiveness of each additional context, we combine these contexts with all of the other contexts for all of the other datasets and evaluate each combination. As all of these hierarchical contexts are evaluated for each dataset, the objective is to measure the effectiveness of *in-context* and *out-of-context* word lists for a dataset, e.g., Android, the in-context word lists are those corresponding to Android development and the operating system. Out of these, only Android development is project specific whereas the operating system context is more abstractly related to the project, though still less generic than the general software-engineering context. The general software-engineering context is in-context, but not nearly as explicitly related to the dataset as the other two word lists. We divide our results for each section into two: tables having at least one in-context feature and tables having no in-context features.

Table VII. Accuracy and Kappa scores using different contextual feature combinations for the Eclipse dataset by using C4.5 decision tree algorithm.

In-context				
Project-specific context	Domain-specific context	General context	Accuracy	Kappa
Eclipse documentation	IDE	Pressman	93.03	0.780
Eclipse documentation	Operating system	Pressman	92.87	0.775
Eclipse documentation	Office	Pressman	92.78	0.771
Eclipse documentation	Web	Pressman	92.86	0.774
Eclipse documentation	IDE	SWEBOK	**93.18**	**0.784**
Eclipse documentation	Operating system	SWEBOK	92.83	0.774
Eclipse documentation	Office	SWEBOK	93.05	0.780
Eclipse documentation	Web	SWEBOK	93.00	0.779
Out-of-context				
Project-specific context	Domain-specific context	General context	Accuracy	Kappa
Android development	Operating system	Pressman	92.62	0.770
Mozilla documentation	Web	Pressman	92.58	0.769
OpenOffice documentation	Office	Pressman	92.34	0.759
Android development	Operating system	SWEBOK	92.61	0.769
Mozilla documentation	Web	SWEBOK	92.67	0.772
OpenOffice documentation	Office	SWEBOK	**92.68**	**0.772**
Best performing features from our previous work [1]				
Project-specific context	Domain-specific context	General context	Accuracy	Kappa
LDA	-	-	**92.90**	**0.775**
Eclipse documentation	-	Pressman	92.81	0.772
Eclipse documentation	-	-	92.86	0.775

4.1. Android Bug Reports

For the Android dataset, the domain specific context comes from the operating system (Linux) feature lists, and the project-specific context comes from the Android development feature lists.

The results are summarized in Table VI. The out-of-context features perform worse than the in-context features. The SWEBOK context features perform at par with the general software-engineering contextual features from the Pressman textbook. The table containing operating system and Android development contexts, which are directly relevant with Android, performs the best. These directly applicable features combined with the Pressman context perform best with an accuracy score of 92.50%, followed closely by the in-context features with SWEBOK.

The contextual tables containing the project-specific context, Android development performed better than the tables with the out-of-context contextual features consisting mainly of general software-engineering context, as expected. The added context produces a marginal improvement in the best performing features from our previous work [1], though still performing lower than the labelled LDA features, albeit marginally. Our table contains 36 contextual features, 10 from Android development, 13 from the operating system, and 13 from Pressman's book, while the labelled LDA approach uses 72 features from 72 word lists.

For Android **RQ1**, **RQ2**, **RQ3** we can see that three in-context contexts produce the best performance; from Table X we can see that each context does provide some value over just BMF-based text comparisons.

4.2. Eclipse Bug Reports

The three hierarchical contexts for Eclipse are general software engineering, IDE, and Eclipse documentation. The in-context features are Eclipse documentation and IDE. Eclipse documentation is project specific, whereas the IDE context is domain-specific.

The results are summarized in Table VII. The out-of-context features perform worse than the in-context features. The SWEBOK context features perform marginally better than the contextual features from pressman textbook context. The table containing all the features that are directly relevant with Eclipse, IDE and Eclipse documentation, perform the best. These directly applicable features with SWEBOK context clocked the highest accuracy, coming in at 93.18%, followed closely by the in-context features with context from the Pressman book.

The contextual tables containing the project-specific context, performed better than the tables with the out-of-context contextual features, *i.e.*, in-context features from other projects. The new context augmented with the domain-specific context produces a marginal improvement in the best performing features from our prior work [1], performing better than the LDA features. Our contextual table contains 48 contextual features, 19 from Eclipse documentation, 14 from IDE, and 15 from SWEBOK book, while LDA uses 20 features.

For Eclipse **RQ1**, **RQ2**, and **RQ3**, we can see that three in-context contexts produce the best performance. However, Table X shows that two contexts can provide nearly equivalent performance.

4.3. Open Office Bug Reports

The three hierarchical contexts for the Open Office dataset are general software engineering, office-book, and Open Office documentation. The in-context features are Open Office documentation and office software [28].

The results are summarized in Table VIII. The out-of-context features perform worse than the in-context features. The Pressman context features perform marginally better than contextual features from the SWEBOK textbook. The table containing all the features that are directly applicable to Office, office software domain specific and Open Office documentation, perform the best. These features combined with Pressman context clocked the highest accuracy of 91.57%.

The contextual tables containing the project-specific context, Office documentation performed better than the tables with out-of-context contextual features, *i.e.*, in-contexts from the other projects, as expected. The new context padded with Office text-book domain-specific context produces a very marginal improvement in the best performing features from our previous work [1], performing better than the LDA features.

For OpenOffice **RQ1**, **RQ2**, and **RQ3**, we can see that three in-context contexts produce the best performance. For OpenOffice the project documentation seems especially strong.

4.4. Mozilla Bug Reports

The three hierarchical contexts for the Mozilla dataset are general software engineering, web development, and Mozilla documentation. The in-context features are Mozilla documentation and web development. Mozilla documentation constitutes the project-specific context, while web development serves as the domain-specific context.

The results are summarized in Table IX. As can be observed, the out-of-context features perform worse than the in-context features. Pressman context features perform at par with contextual features from SWEBOK textbook. The table containing all the features that are directly relevant with Mozilla, Web development and Mozilla documentation, perform the best. These features combined with Pressman context performed with the highest accuracy of 93.07%.

Mozilla documentation performed better than the tables with out-of-context contextual features, *i.e.*, in-contexts from the other projects. The new context, augmented with Web development domain-specific context, produces a marginal improvement compared to the best performing features from our previous work [1], but performs marginally worse than the LDA features.

For Mozilla **RQ1**, **RQ2**, and **RQ3**, all three contexts matter, which is also evident in the next section.

4.5. The Effects of Context

We seek to understand the effect of context on model performance. In this section, we re-run the experiments of the prior section 10 times on contexts relevant to the projects themselves. We then

Table VIII. Accuracy and Kappa scores using different contextual feature combinations for the Open Office dataset by using C4.5 decision tree algorithm.

In-context				
Project-specific context	Domain-specific context	General context	Accuracy	Kappa
OpenOffice documentation	Office	Pressman	**91.57**	**0.728**
OpenOffice documentation	IDE	Pressman	91.38	0.721
OpenOffice documentation	Operating system	Pressman	91.37	0.722
OpenOffice documentation	Web	Pressman	91.51	0.723
OpenOffice documentation	Office	SWEBOK	91.42	0.724
OpenOffice documentation	IDE	SWEBOK	91.36	0.721
OpenOffice documentation	Operating system	SWEBOK	91.23	0.719
OpenOffice documentation	Web	SWEBOK	91.32	0.720
Out-of- context				
Project-specific context	Domain-specific context	General context	Accuracy	Kappa
Eclipse documentation	IDE	Pressman	90.82	0.707
Mozilla documentation	Web	Pressman	**91.22**	**0.716**
Android development	Operating system	Pressman	90.94	0.710
Eclipse documentation	IDE	SWEBOK	90.75	0.703
Mozilla documentation	Web	SWEBOK	91.17	0.715
Android development	Operating system	SWEBOK	90.93	0.709
Best performing features from our previous work [1]				
Project-specific context	Domain-specific context	General context	Accuracy	Kappa
LDA	-	-	90.71	0.699
OpenOffice documentation	-	Pressman	**91.51**	**0.728**

Table IX. Accuracy and Kappa scores using different contextual feature combinations for the Mozilla dataset by using C4.5 decision tree algorithm.

In-context				
Project-specific context	Domain-specific context	General context	Accuracy	Kappa
Mozilla documentation	Web	Pressman	**93.07**	**0.779**
Mozilla documentation	IDE	Pressman	92.84	0.771
Mozilla documentation	Operating system	Pressman	92.83	0.771
Mozilla documentation	Office	Pressman	92.83	0.771
Mozilla documentation	Web	SWEBOK	92.96	0.775
Mozilla documentation	IDE	SWEBOK	92.82	0.770
Mozilla documentation	Operating system	SWEBOK	92.88	0.772
Mozilla documentation	Office	SWEBOK	92.93	0.773
Out-of-context				
Project-specific context	Domain-specific context	General context	Accuracy	Kappa
Android development	Operating System	Pressman	92.72	0.769
Eclipse documentation	IDE	Pressman	92.69	0.768
OpenOffice documentation	Office	Pressman	92.71	0.769
Android development	Operating System	SWEBOK	92.75	0.769
Eclipse documentation	IDE	SWEBOK	92.70	0.768
OpenOffice documentation	Office	SWEBOK	**92.77**	**0.770**
Best performing features from our previous work [1]				
Project-specific context	Domain-specific context	General context	Accuracy	Kappa
LDA	-	-	**93.14**	**0.780**
Eclipse documentation	-	Pressman	92.89	0.772

Table X. Only in-context contexts per each project. This table shows how adding contexts tends to slowly improve the performance of the machine learner. "-" indicates the context is not used. Classified with a C4.5 Decision Tree Algorithm using 10 fold cross validation.

In-Context					
Project	Project Context	Domain Context	General Context	Accuracy	Kappa
Android	-	-	-	89.55	0.6589
Android	Android Documentation	-	-	92.05	0.7495
Android	-	Operating System	-	91.61	0.7353
Android	-	-	Pressman	92.07	0.7514
Android	-	-	SWEBOK	91.52	0.7306
Android	Android Documentation	Operating System	-	92.33	0.7584
Android	Android Documentation	-	Pressman	92.32	0.7577
Android	-	Operating System	Pressman	92.17	0.7541
Android	Android Documentation	-	SWEBOK	92.06	0.7487
Android	-	Operating System	SWEBOK	91.97	0.7456
Android	Android Documentation	Operating System	Pressman	92.38	0.7598
Android	Android Documentation	Operating System	SWEBOK	92.20	0.7541
Eclipse	-	-	-	91.32	0.7354
Eclipse	Eclipse Documentation	-	-	92.62	0.7681
Eclipse	-	IDE	-	92.69	0.7707
Eclipse	-	-	Pressman	92.47	0.7639
Eclipse	-	-	SWEBOK	92.63	0.7680
Eclipse	Eclipse Documentation	IDE	-	92.87	0.7751
Eclipse	Eclipse Documentation	-	Pressman	92.81	0.7732
Eclipse	-	IDE	Pressman	92.66	0.7683
Eclipse	Eclipse Documentation	-	SWEBOK	92.86	0.7743
Eclipse	-	IDE	SWEBOK	92.78	0.7715
Eclipse	Eclipse Documentation	IDE	Pressman	92.86	0.7749
Eclipse	Eclipse Documentation	IDE	SWEBOK	92.95	0.7778
Mozilla	-	-	-	91.27	0.7233
Mozilla	Mozilla Documentation	-	-	92.75	0.7677
Mozilla	-	Web	-	92.82	0.7695
Mozilla	-	-	Pressman	92.74	0.7663
Mozilla	-	-	SWEBOK	92.63	0.7626
Mozilla	Mozilla Documentation	Web	-	92.90	0.7722
Mozilla	Mozilla Documentation	-	Pressman	92.90	0.7729
Mozilla	-	Web	Pressman	92.86	0.7711
Mozilla	Mozilla Documentation	-	SWEBOK	92.79	0.7690
Mozilla	-	Web	SWEBOK	92.79	0.7687
Mozilla	Mozilla Documentation	Web	Pressman	92.91	0.7731
Mozilla	Mozilla Documentation	Web	SWEBOK	92.87	0.7722
OpenOffice	-	-	-	88.92	0.6451
OpenOffice	OpenOffice Documentation	-	-	91.44	0.7238
OpenOffice	-	Office	-	90.98	0.7069
OpenOffice	-	-	Pressman	90.89	0.7056
OpenOffice	-	-	SWEBOK	91.04	0.7102
OpenOffice	OpenOffice Documentation	Office	-	91.46	0.7245
OpenOffice	OpenOffice Documentation	-	Pressman	91.38	0.7215
OpenOffice	-	Office	Pressman	91.23	0.7167
OpenOffice	OpenOffice Documentation	-	SWEBOK	91.31	0.7205
OpenOffice	-	Office	SWEBOK	91.25	0.7173
OpenOffice	OpenOffice Documentation	Office	Pressman	91.43	0.7234
OpenOffice	OpenOffice Documentation	Office	SWEBOK	91.45	0.7248

apply factor analysis to determine if the different kinds of context matter. We apply these tests to all of the projects' kappa and accuracy performance at once.

Table X depicts the median performance of 10 repeated 10-cross-folds validations – each with a different cross-fold split random seed – over the in-context contexts of each project. This results in 4800 model evaluations (10-folds, 480 times) over relevant contexts, resulting in 480 accuracy and kappa values. In all cases we can see that adding context improves both accuracy and kappa values (versus not adding context). This difference is significant according to the Student's T-Test ($p < 2 \times 10^{-16}$). Individually evaluating each combination of contexts (general, domain-specific, and project-Specific) we find that all relevant context combinations have statistically significant performance, better than no context at all.

We applied ANOVA factor analysis on contexts versus kappa performance. We found positive and significant effects ($p < 3 \times 10^{-5}$) on performance given general, domain-specific, and project-specific contexts, including interactions between general, domain-specific, and project-specific contexts. By applying the *Tukey Honest Significant Differences* (Tukey HSD) test we observed the 95% confidence intervals, related to the effect of having general contexts, domain-specific contexts, or project-specifics contexts on kappa scores, were $[0.011, 0.020]$, $[0.0091, 0.018]$, and $[0.013, 0.022]$ respectively, with positive mean effect all greater than 0.013 ($p < 2 \times 10^{-16}$). This implies that that the introduction of relevant in-context contexts, regardless of the level of context, tends to have a positive interaction on Kappa scores. Similar behaviour is observed for accuracy scores.

Thus for **RQ1**, **RQ2**, **RQ3** all three levels of context – project-specific, domain-specific, and general – help deduplicate bug reports. None of them hinder deduplication.

The general context and domain-specific context seem to interact more negatively, as the difference between the combination of general contexts combined with domain contexts versus just general contexts is not statistically significant after adjustment. The significant interactions tend to be context versus lack of context. So interactions in general as significant as per the ANOVA, but the Tukey HSD does not find the majority of comparisons within an interaction to be significant. A two-context interaction was significant: general context combined with project-specific context had a statistically significantly kappa difference: 0.007 with $p < 0.028$. The same was a true for accuracy: 0.232 with $p < 0.016$. Of the 3 context interactions, all 3 contexts together – general, domain-specific, and project-specific contexts – had a statistically significant difference ($p < 0.025$) of 0.013 for kappa performance, similar accuracy performance: 0.39 with $p < 0.012$.

Thus for **RQ4** we can see that having more than one, preferably three, in-context contexts helps deduplication performance in terms of accuracy and kappa scores.

If we look at the cosine-similarity between contexts of duplicate and non-duplicate bug-report pairs, we find that, for Eclipse, OpenOffice, and Android, the distributions are significantly different according to the Wilcoxon rank sum test ($p < 2 \times 10^{-6}$). Yet for Mozilla, many of the duplicate and non-duplicate cosine differences are not statistically significant. Overall the median cosine-similarity of duplicate pairs is different than those of non-duplicate pairs according to a Wilcoxon ($p < 0.0056$) applied across projects. This is interesting because it means that there's a difference between median cosine-similarity of duplicate pairs, and the cosine pairs tend to have larger cosine-similarities.

In some of the tables SWEBOK and Pressman produce different levels of performance; are these significant? A Wilcoxon rank sum test of Kappa scores between SWEBOK and Pressman contexts suggests that the differences in performance are not significant with a p-value of 0.5361. TukeyHSD suggests that the 95% confidence interval of the difference between SWEBOK and Pressman is $[-0.010, 0.005]$ with a insignificant p-value of 0.77.

Thus we conclude that relevant context matters: context positively improves kappa and accuracy performance, and three contexts are better than just one in terms of kappa and accuracy performance, across the projects. Based on these results, we believe the most powerful contexts are the general contexts and the project specific contexts, our domain-specific contexts only seemed effective in combination with other contexts. Analysis of cosine-similarity shows that cosine-similarity between contexts is a relevant feature in deciding if a pair is a duplicate or not.

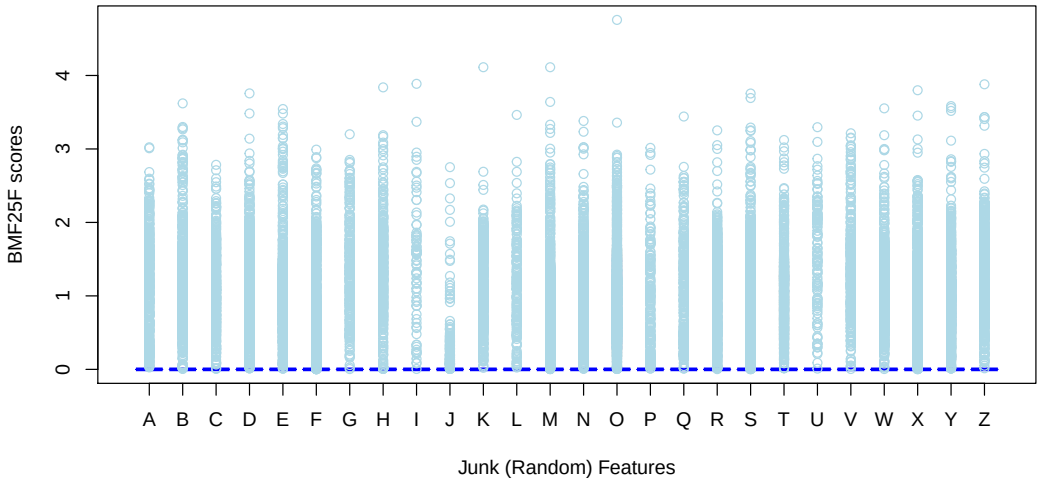

Figure 3. BM25F scores over random English words context (Junk context).

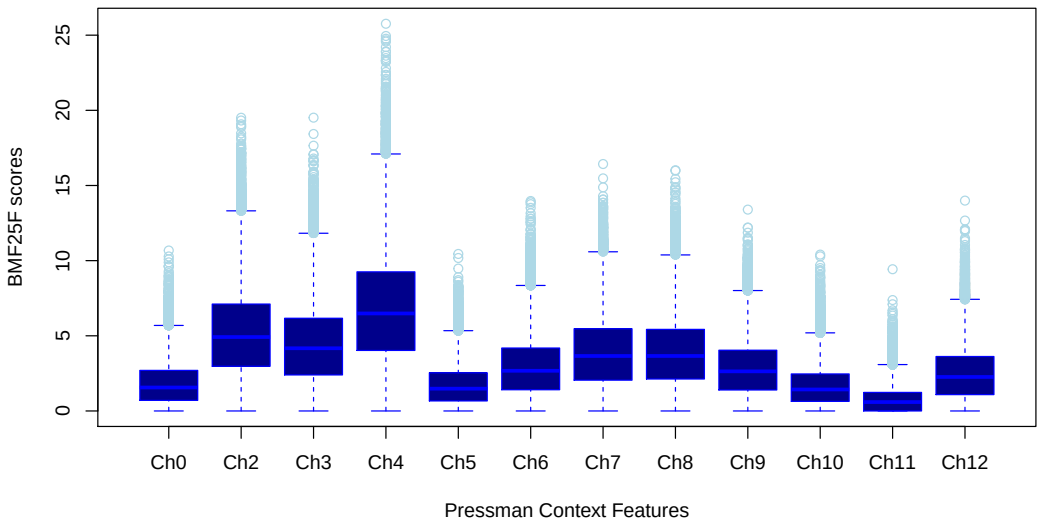

Figure 4. BM25F scores over Pressman General SE context.

4.6. BM25F scores distributions over contexts

We present the distribution of BM25F scores of five contexts over Open Office bug reports. We take four contexts—General Software Engineering (Pressman and SWEBOK), Office domain-specific context, and Office project-specific context. As a baseline, we take an additional context—random context generated by randomly created A-Z words lists of English words, same as used by Alipour *et al.* [3, 2]. The plots showing the BM25F scores over the Office bug reports with these five contextual word lists are shown in Figures 3, 4, 5, 6, and 7. As can be observed in Figure 3, random context

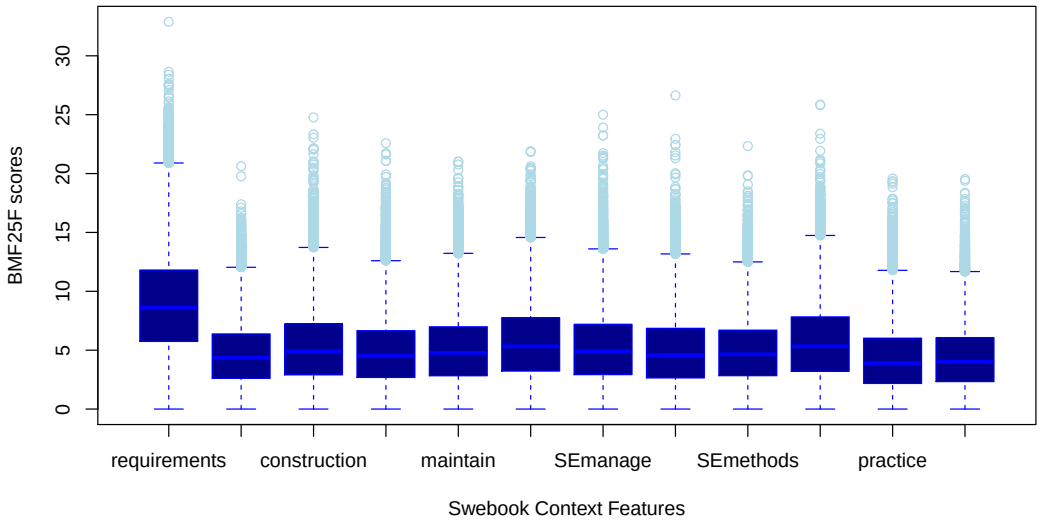

Figure 5. BM25F scores over SWEBOK General SE context.

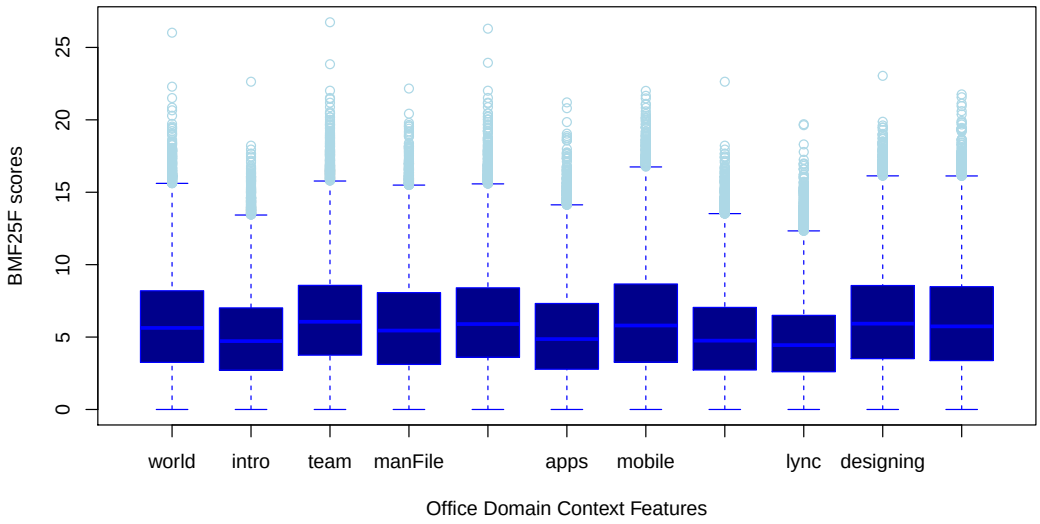

Figure 6. BM25F scores over Office domain context.

has very low BM25F similarity scores with bug reports, in-fact nearly zero. The project-specific, Office documentation context has very high BM25F scores, much more than other contexts. Within the General Software Engineering context, the Pressman book has, on average, lower scores than SWEBOK. As we can see from the results in the previous section, the SWEBOK context performs better than Pressman on the OpenOffice Corpus, though only marginally. The Office domain-specific context has much lower BM25F scores on average over the features than OpenOffice documentation

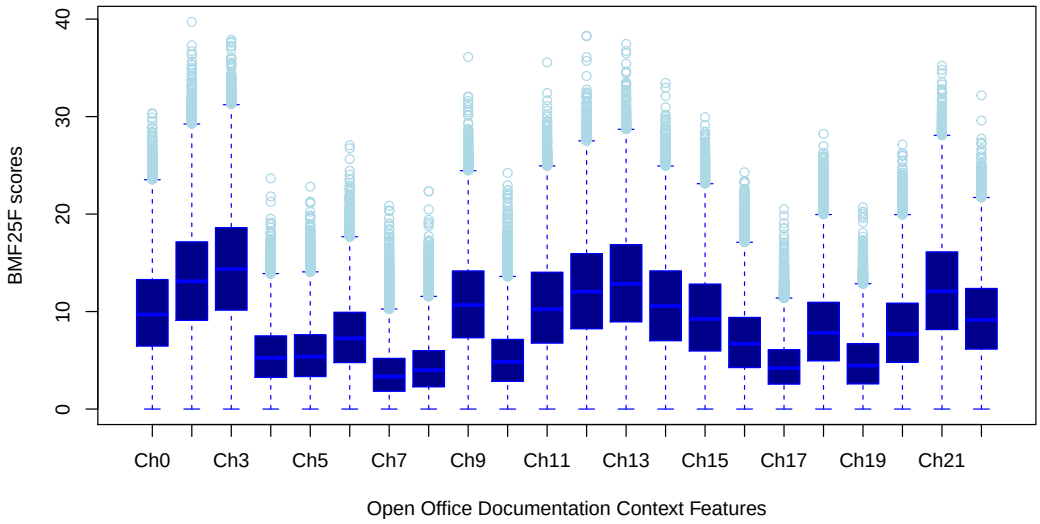

Figure 7. BM25F scores over OpenOffice documentation context.

context, though a bit higher than General Software Engineering contexts. These visualizations show that these different contexts add information to the deduplication task, since different contextual features have overlaps with the different bug reports. They do not add any random information like the one demonstrated in the case of random context, having minimal overlap with the Office bug reports. Similar trends are observed in other datasets as well.

Summary As can be observed, in all the four bug-report datasets, combinations of in-context features performed quite well, whereas the out-of-context features performed worse, despite being combined with general software-engineering contextual features. It is important to note, though, the differences in performance of various contextual feature sets is between 1-2% but since our dataset consists of 80% non-duplicates and 20% duplicates, the 1-2% additional gain in accuracy should be viewed as 5-10% gain in detecting pair of duplicates using these features. The best performing classifier, C4.5 Decision Tree, has unigram similarity at the top of the tree over all the experiments, implying that textual similarity between the bug reports is the most important factor in deciding whether a pair of bugs reports represent a duplicate bug. The contextual features come next, providing additional cues to classify as duplicate bugs or not. Previous works [3, 2, 1] have shown that additional contextual features capturing similarity of bug reports with the contextual word lists provides gains over the popular textual similarity features.

Our hierarchy of contexts, with added layer of domain specific context, performs better than our only-project specific, or combination of project-specific and general software-engineering contexts, used in our previous study [1]. SWEBOK features, added for additional general software-engineering literature context perform similarly to the Pressman features. Alipour *et al.* [3, 2] showed how adding context (contextual word lists as features) to the textual bug-deduplication methods can drastically improve the performance of detection. Our previous work [1] demonstrated how we can use software-engineering literature for building that context. This work examines the software-engineering literature context further by adding more abstracted contexts, as well as validates our previous work by showing a contrast with out of context features.

5. THREATS TO VALIDITY

Construct validity is threatened by the literature sources selected for extracting the word lists and the word-list topics selected, that is, the topics according to which the chapters in the literature were labelled. Labelling could have been improved by multiple round and multiple author coding/labelling process to ensure agreement of labels. Labelling could also suffer from correctness as chapters and documentation might not be as on-topic as the authors hope. Another threat is the word list of English words used to filter the frequent words to only use contextual words for the similarity score calculation. Construct validity is further hampered the sources of data, as the correctness of the duplicate bug report labels or the lack of duplicate bug report labels depends on the developers who marked the bug reports that way.

External validity is threatened by our choice of four platforms used here. We try to address it using these domain-diverse set of platforms, which have a long history of bug-tracking systems.

6. DISCUSSION

In this study we have built upon our "*Software-literature context method*" for capturing additional contextual space with the hierarchy of contexts. Besides the hierarchy of contexts, we also use additional context from SWEBOK to incorporate contextual features at a general software engineering level. The combined hierarchy of contextual features performed better than all the previous contextual features that were exploited in our previous study [1]. We also demonstrate the effectiveness of domain-specific features by contrasting their efficacy with less specific, cross-domain features from other datasets. The hierarchy of contextual features extracted using our method fared either better or at-par than the LDA contexts, but fared slightly worse than the labelled-LDA approaches. As labelled-LDA is an extremely time-intensive approach, we consider that a reasonable trade off.

In the case of the Android data set, the three layered hierarchical context features performed slightly worse than labelled-LDA features, though better than our previous features. However, the *software-literature context* features took much less time to produce, only half a person-hour compared to 60 person-hours taken to create labelled LDA lists, while suffering only a minor loss in accuracy. Labelled LDA features are labour intensive and not available for all projects such as Eclipse, Mozilla, and Open Office. Alipour *et al.* used unsupervised LDA for these datasets. Our *software-literature context method* performs better than LDA across the Eclipse and Open Office datasets, while performing at par on the Mozilla dataset. As established in previous work [1], the features extracted using *software-literature context method* are simple, general, easy to extract, and share, as compared to LDA approaches. LDA requires extraction of all the bug descriptions, knowledge of how to use rather sophisticated tools, and requires that the parameters be appropriately tuned. Additionally, for both variants of LDA, the time and resources required increases over time as the lists need to be updated when the number of bug reports in the bug-tracker system increases.

In contrast, our method requires a very simple chapter by chapter labelling of (contextual) software literature such as books, guides, or documentation, then, a simple conversion to word lists that can be done easily using command line. The method for constructing BM25F similarity scores is same for both LDA and our method. The output word lists are easily share-able across the projects. As well, these word lists do not require any updates over time across the whole corpus to extract latent topics unlike LDA. For example, word lists for OS can be shared across open source OS projects. There is also potential to use these lists to detect bugs inherited by source code reuse.

The domain-specific context provides additional gains in the accuracy scores, and consistently performs better than the out-of-context features used for cross-domain analysis. The major advantage of domain-specific context is that it can be applied across a variety of projects. For example, the operating-system context can be applied across projects related to Android, Linux, BSD, and Sun OS. The developers can share these domain-specific features across their projects in the same domain, from the relevant software literature.

The process involves labelling chapters from software-literature sources and extracting word lists with simple tokenization. The word lists extracted from general software-engineering literature, are relevant across all software projects and performed consistently well along with other software project-specific features or project-domain features. In our previous work [1], we had demonstrated the utility of the general software-engineering features with just one literature source from the Pressman book. In this study, added features from the SWEBOK, the industry accepted guide to software-engineering practices, has been used to reinforce our previous conclusion.

In this work we also demonstrated the effects of using an out-of-context contextual feature hierarchy, *i.e.*, features not directly related to the project on our datasets. These features performed consistently worse than the feature hierarchies containing the project-specific features. This reinforces our observations in previous work on the superiority of contextual features. The performance of the generic software-engineering literature features, demonstrated with two literature sources, Pressman and SWEBOK, suggests that even higher-level contexts, that are not specific to the project domain but rather to the general software domain, provide a useful and reusable context, effective for bug-report deduplication. Furthermore, higher-level contexts can be freely shared and reused by practitioners with little or no effort compared to extracting features from domain-specific texts, or using LDA.

7. CONCLUSIONS

Our work establishes a method to improve the detection of duplicate bug reports using hierarchical contextual information extracted from software-engineering textbooks, project-related software literature, and project documentation. The method is an extension of our previous work [1], which introduced the *software-literature context method*, a method of using automatically generated contextual features to identify and deduplicate bug reports.

We show the effectiveness of more specific contextual features by comparing the results from using in-context features, such as project-specific documentation, with the results from using out-of-context features, such as project-specific context from other datasets. We refer to the differing levels of abstraction as a hierarchy of contexts; the lower the context is on the hierarchy, i.e., the more directly related the context is, the better performance we find in our classification results.

Compared to our previous work, we find that the domain-specific context provides additional gains in accuracy, while being able to be applied across a variety of projects with no additional processing. For instance, the operating-system contextual features can be applied across projects based on the Linux kernel, such as Android, all Linux distributions, BSD, and Sun OS. This general applicability enables us to eliminate our dependence on project-specific word lists that must be generated for each project, and to further decrease the labour required to deduplicate bug reports.

This enhances our previous study by showing the contrast between our semi-automated method with the LDA-based approaches, which are project-specific and need to be updated manually to incorporate data from new bug reports. This new method, by comparison, does not need to be updated to incorporate new bug reports. When new bug reports arrive, the only work that needs to be done is to calculate the feature values for the new bug reports. The improved performance on all four of the datasets that we use demonstrates the utility of these domain-specific contextual features, which, though much less generic than the general software-engineering contextual features, can still be shared across a number of projects in the same domain.

As well, by using contextual features generated from SWEBOK in addition to Pressman, our study reinforces the superiority of general software-engineering features. The extremely general nature of the features, general software-engineering and domain-specific features, means that they can be used across a wide array of software projects, as demonstrated on the four diverse datasets that we test the features on. The terms generated from the general software-engineering literature are used across all software platforms, and hence the contextual features generated from this literature are applicable across all software platforms. Evaluating the generic features alongside the additional contextual features shows the robustness of our software-literature contextual method.

This paper reinforces the *software-literature context method*, proposed in our previous work, and demonstrates the importance and effectiveness of domain in the task of automating bug deduplication, in order to reduce manual effort. We confirm clearly through factor analysis that project documentation (**RQ1**), domain-specific context (**RQ2**), and general contexts (**RQ3**) help deduplicate bug reports, but also that the interaction of these contexts significantly improves deduplication performance (**RQ4**).

In this work, we used four bug-tracking systems—diverse platforms having a long history of development. The method can be used on bug-tracking systems of freely available platforms such as Linux, Debian, and Apache as well. Future directions include automating the process of context extraction, exploiting ontological relationships, and comparison of different representations of context.

REFERENCES

1. Karan Aggarwal, Tanner Rutgers, Finbarr Timbers, Abram Hindle, Russ Greiner, and Eleni Stroulia. Detecting duplicate bug reports with software engineering domain knowledge. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 211–220. IEEE, 2015.
2. Anahita Alipour. A contextual approach towards more accurate duplicate bug report detection. Master's thesis, University of Alberta, Fall 2013.
3. Anahita Alipour, Abram Hindle, and Eleni Stroulia. A contextual approach towards more accurate duplicate bug report detection. In *Proceedings of the Tenth International Workshop on Mining Software Repositories*, pages 183–192. IEEE Press, 2013.
4. W Appel Andrew and P Jens. Modern compiler implementation in java, 2002.
5. Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful really? In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 337–345. IEEE, 2008.
6. Pierre Bourque, Richard E Fairley, et al. *Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.
7. Chris Buckley and Gerard Salton. Stop word list, December 2013.
8. Randy Connolly and Ricardo Hoar. *Fundamentals of Web Development*. Pearson Higher Ed, 2015.
9. Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 1084–1093. IEEE Press, 2012.
10. Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. Understanding android fragmentation with topic analysis of vendor-specific bugs. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 83–92. IEEE, 2012.
11. Geoffrey Holmes, Andrew Donkin, and Ian H Witten. Weka: A machine learning workbench. In *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*, pages 357–361. IEEE, 1994.
12. Nicholas Jalbert and Westley Weimer. Automated duplicate detection for bug tracking systems. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 52–61. IEEE, 2008.
13. Adam Kiezun. Basic tutorial eclipse 3.1.
14. Nathan Klein, Christopher S Corley, and Nicholas A Kraft. New features for duplicate bug detection. In *MSR*, pages 324–327, 2014.
15. Alina Lazar, Sarah Ritchey, and Bonita Sharif. Improving the accuracy of duplicate bug report detection using textual similarity measures. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 308–311. ACM, 2014.
16. Robert Love. *Linux kernel development*. Pearson Education, 2010.
17. Mozilla Developer Network and individual contributors. Mozilla developer guide.
18. Mark Lawrence Murphy. *The Busy Coder's Guide to Advanced Android Development*. CommonsWare, LLC, 2009.
19. Anh Tuan Nguyen, Tung Thanh Nguyen, Tuan N Nguyen, Daniel Lo, and Chengnian Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 70–79. IEEE, 2012.
20. Roger S Pressman and Waman S Jawadekar. Software engineering. *New York 1992*, 1987.
21. Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 499–510. IEEE, 2007.
22. Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 253–262. IEEE Computer Society, 2011.
23. Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 45–54. ACM, 2010.
24. Sun Microsystems. Openoffice.org 3.0 developer's guide, 2008.

25. Ashish Sureka and Pankaj Jalote. Detecting duplicate bug report using character n-gram-based features. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 366–374. IEEE, 2010.
26. Ferdian Thung, Pavneet Singh Kochhar, and David Lo. Dupfinder: integrated tool support for duplicate bug report detection. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 871–874. ACM, 2014.
27. Yuan Tian, Chengnian Sun, and David Lo. Improved duplicate bug report identification. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 385–390. IEEE, 2012.
28. Kevin Wilson. Microsoft office 365. In *Using Office 365*, pages 1–14. Springer, 2014.