Detecting Duplicate Bug Reports with Software Engineering Domain Knowledge

Karan Aggarwal, Tanner Rutgers, Finbarr Timbers, Abram Hindle, Russ Greiner, and Eleni Stroulia
Department of Computing Science
University of Alberta
Edmonton, Canada
{kaggarwa, trutgers, timbers, abram.hindle, rgreiner, stroulia}@ualberta.ca

***Abstract*—In previous work by Alipour *et al.*, a methodology was proposed for detecting duplicate bug reports by comparing the textual content of bug reports to subject-specific contextual material, namely lists of software-engineering terms, such as non-functional requirements and architecture keywords. When a bug report contains a word in these word-list contexts, the bug report is considered to be associated with that context and this information tends to improve bug-deduplication methods.**

In this paper, we propose a method to partially automate the extraction of contextual word lists from software-engineering literature. Evaluating this *software-literature context method* on real-world bug reports produces useful results that indicate this semi-automated method has the potential to substantially decrease the manual effort used in contextual bug deduplication while suffering only a minor loss in accuracy.

***Index Terms*—duplicate bug reports; information retrieval; software engineering textbooks; machine learning; software literature; documentation.**

I. Introduction

An integral part of the quality-assurance processes for many modern software projects is the use of issue-tracking systems; these systems record bug reports or, issues that developers, testers, and users encounter for a particular software system. In effect, these systems serve as repositories of bug reports, stack traces, and feature requests. Issue tracker constitute a proxy for measuring the developers' productivity based on their progress in correcting or addressing an issue or bug report. Bug reports are usually written in natural-language text, which implies that the same issue can potentially be described in different ways by the various users or developers that encounter this issue. Typically the vocabulary used by developers differs from that used by users; thus, to recognize that two bug reports refer to the same bug, a triager, often an experienced developer, has to use their expert knowledge to translate bug reports into a technical language that developers can understand. Often these reports are duplicates of existing bug-reports, that have been or are currently being addressed. Recognizing duplicate reports is an important problem that, if solved, would enable developers to fix bugs faster, and prevent them from wasting time by addressing the same bug multiple times.

To reduce manual triaging effort, considerable research has been done to find an automated method of detecting duplicate bugs. Prior work from Runeson *et al.* [1] and Sun *et al.* [2], [3] in *bug report de-duplication*, detection of duplicate bug reports, measures bug report similarity by combining natural language similarity measures of bug report descriptions with categorical bug attributes such as "component", "type", and "priority". Typically these approaches use off-the-shelf document-similarity measures and apply them to bug reports. While this is effective, it ignores an important context: bug reports are not New York Times articles, bug reports are about software projects.

Recognizing this important fact, Alipour *et al.* [4], [5] exploited the software-engineering and project-specific context to boost bug-report deduplication. By exploiting contextual data, comparing a bug report to terms referring to non-functional requirements or architectural descriptions, Alipour *et al.* improved bug-deduplication performance of Sun *et al.* [3]. This contextual method used manually created 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 effort-intensive process [6]. Alipour concluded that contextual features tend to reveal the relationship between the bug report text and concepts such as non-functional requirements or architectural modules. These relationships can be exploited in bug report deduplication when different terminology is used to describe the same scenario. Thus not only could one compare text between bug reports, one could compare their contexts and associations as well.

The method presented in this paper, called the *software-literature context method*, exploits context and further reduces the manual effort associated with detecting duplicate bug reports with generic contextual features extracted from software-engineering literature, rather than project-specific analysis. These features are general enough to apply to any software-development project. Thus, bug reports are compared using natural-language similarity measures, such as BM25F, to word lists extracted from the chapters of two software-engineering textbooks – Pressman's "Software Engineering: A Practitioner's Approach" [7] and "The Busy Coder's Guide to Android Development" [8] – and documentation of three open-source projects: Eclipse, Mozilla, and Open Office. Our method is different from the method described by Alipour *et. al.* as it uses word lists extracted from software literature instead of using bug reports from the software projects to extract the word lists. These software-literature context word lists reflect the software-development processes, and the evolution of project, and hence depict the bug report descriptions. The software-engineering book by Pressman describes these processes, while the Android book and the project documentation are used as technical manuals by developers working on those projects. Hence, these texts are quite relevant to the task of bug duplication detection.

In this paper, we use four bug datasets from open-source projects, Android, Open Office, Mozilla, and Eclipse (containing approximately $37,000$, $42,000$, $72,000$, and $29,000$ bug reports correspondingly) and we follow the bug-deduplication methodology of Alipour *et al.*

Our results are similar to the results reported by Alipour *et al.* but our method requires far less time and effort and is more general and easy to share.

- Documentation and software-engineering textbook features were 1.48% less accurate on Android dataset than Labelled LDA features but these features took the authors only 0.5 hours to extract, while the labelled LDA features took 60 hours to extract due to annotation.
- In the cases of the Eclipse, Mozilla and Open Office datasets, our method performs on par with an unsupervised LDA method that requires around one person-hour for word list generation and extraction of all the bug descriptions from the bug reports in the repository. Hence, we conclude that the *software context literature method* is much more cost effective.
- Furthermore, the *software context literature method* is more robust as the word lists can be extracted from available authoritative sources, which can be chosen to be directly relevant to the code repositories examined, and automatically kept up to date. Conversely, the labelled LDA word lists requires manual annotation of bug reports with functional features and requires regular manual updates as the bug repository evolves.
- Unsupervised LDA can be used and has been used by others [9], yet it requires the time to run LDA across all bug-reports and the tuning of the LDA parameters is a cumbersome process, especially as the number of bug reports grows.
- Finally, this method is easy to use, as developers only need to use already extracted word-lists, or need to label textbook and project documentation chapters to extract the word lists. The *software-literature context method* benefits from generalizability and ease of share of contextual word-lists, implying that if someone shared their extracted word-lists, like the current authors have, the cost of employing such a word-list would be the cost of a download.

The rest of the paper is organized as follows. Section II reviews the relevant literature. Section III describes the evaluation methodology, Section IV presents the results of evaluation, and Section V discusses the results. Finally, future research directions are highlighted in Section VI and the paper is concluded in Section VII.

II. Related Work

Most bug-deduplication methods use textual analysis to detect duplicate bug reports. Runeson *et al.* [1] used NLP techniques to detect 66% of the duplicate reports of Sony Ericson Mobile Communications. Bettenberg *et al.* [10] used machine-learning classifiers to triage the reports based on representing the report titles and descriptions as word vectors. Using support vector machines (SVM) and naïve Bayes, they obtained accuracy scores of roughly 65%.

Jalbert *et al.* [11] used categorical features of bug reports, textual-similarity and graph-clustering techniques to filter out duplicate reports. They developed an automatic method to detect the duplicate bug reports, and allowed only one of the duplicates to reach developers. They evaluated their method on a dataset of $29,000$ bug reports from Mozilla Firefox and were able to filter out 8% of the duplicate bug reports.

Wang *et al.* [12] proposed a method using execution traces on the Eclipse dataset using NLP techniques; however, their methodology relies on the manual extraction of the execution traces, which makes it extremely time-intensive.

Surekha *et al.* [13] used an *n*-gram based textual model on an Eclipse dataset to report top-*k* bug reports that could be duplicate of a given bug report. Building on their work, Sun *et al.* [3] proposed a new model based on the BM25F score that uses the term frequency-inverse document frequency (TF-IDF) score to measure report similarity. In addition, they use categorical features such as priority and severity to produce substantial improvements over previous methods. Sun *et al.* sorted the reports into different "*buckets*" according to the underlying bugs, and focused on sending incoming duplicate reports to the appropriate bucket. They report top-*k* bug reports for a given bug that could be duplicate and use the bugs marked as duplicates of the bug by triager from the original dataset to find an improvement of 10-27% over Surekha *et al.*. One issue with this kind of evaluation is only true-positives are queried. Bug reports with no duplicates are not evaluated or queried, thus true-negatives are not tested.

Alipour [4], [5] improved upon the work of Sun *et al.* [3] by adding contextual features using both labelled and unlabelled LDA generated word lists [6] 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.*. Klein *et al.* [9] and Lazar *et. al.* [14] have leveraged the same dataset against new textual metrics based 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 extraction effort from the client (this paper applies them to 4 different projects).

III. Methodology

This section describes how to measure the effectiveness of using contextual features, such as similarity to software engineering texts, to determine if bug reports are duplicates of each other. First the processes of curation of the datasets and the contextual word-lists used to measure contextual features are described. Next, the extraction of contextual features is discussed. Finally, the evaluation of how well these contextual features help bug-report deduplication is discussed.

The Android, Eclipse, Mozilla, and Open Office bug report dataset used for analysis consists of approximately $37,000$, $47,000$, $72,000$, and $29,000$ bug reports respectively, being same as used by Alipour *et al.* [4], [5]. 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 the years 2008 to 2010. Each report contains the following fields: *Bug ID, description, title, status, component, priority, type, version,*

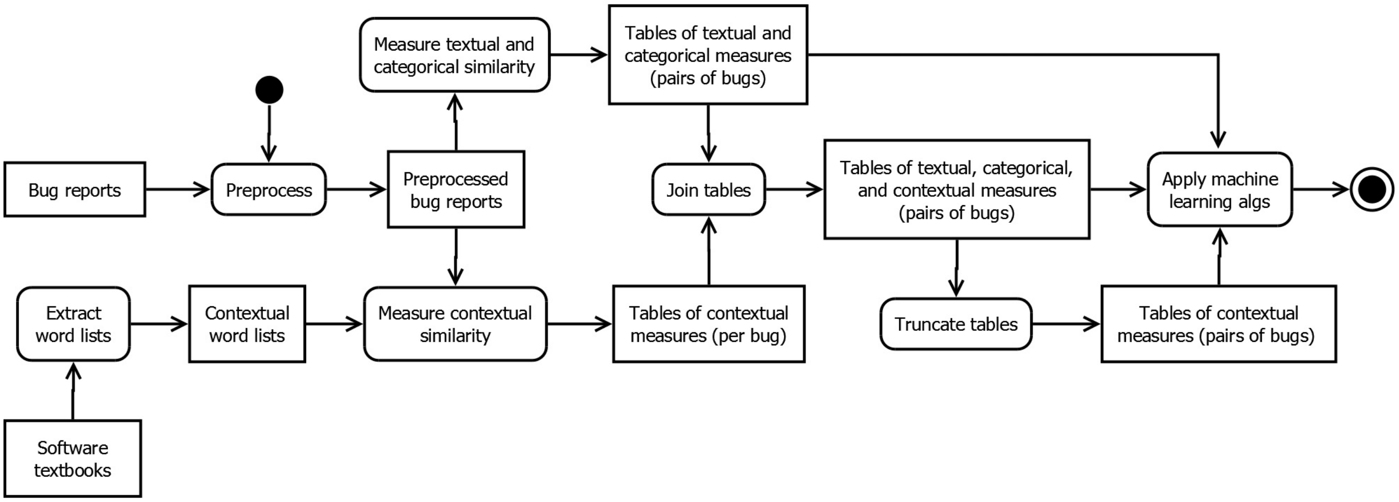
Measure textual and categorical similarity
Tables of textual and categorical measures (pairs of bugs)
Bug reports
Preprocess
Preprocessed bug reports
Join tables
Tables of textual, categorical, and contextual measures (pairs of bugs)
Apply machine learning algs
Extract word lists
Contextual word lists
Measure contextual similarity
Tables of contextual measures (per bug)
Truncate tables
Tables of contextual measures (pairs of bugs)
Software textbooks

Fig. 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.

open date, close date, and *Merge ID*. In case of Mozilla, the name *dup_id* is used in place of MergeId, in the bug reports. If one 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 I 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, therefore its MergeID is empty, whereas Bug `14518` is a duplicate and has a non-empty MergeID assigned to it. Figure I depicts the workflow of the *software-literature context method*.

A. Contextual-Features Extraction

The following 8 word lists were used as context in this study:

1) *General software engineering*: This word list was extracted from Pressman’s textbook [7]. The book was split into 13 different word lists corresponding to its chapters like architecture, UI design, formal methods, and testing. The complete process took around half an hour for a single person.
2) *Android development*: This word list was extracted from the chapters of Murphy [8] to produce ten word lists describing features like widgets, activities, databases. They were all related specifically to Android application development, and the process took around half an hour for a single person.
3) *Eclipse documentation*: This word list was extracted from the Eclipse platform documentation for Eclipse 3.1 [15]. The documentation was split into 19 different word lists relating to using Eclipse, debugging, and IDE features. The process around half an hour for a single person.
4) *Open Office documentation*: This word list was extracted from the developer documentation for Open Office 3.0 [16]. The documentation was split into 22 different word lists relating to using various components like spreadsheets, text documentation, database access, API design, GUI. The process around half an hour for a single person.
5) *Mozilla documentation*: This word list was extracted from the online developer guide for Mozilla [17]. Unlike Eclipse, and Open Office there is no one central documentation for Mozilla products; the online documentation consists of several webpages, with very short descriptions catering to online audience. The documentation was split into 13 different word lists relating to using various components such as browser, Javascript, debugging tools, and testing. The process around half an hour for a single person.
6) *Labelled LDA*: These word lists were extracted using labelled LDA on the Android bug reports by Dan *et al.* [6]. There are 72 word lists on a variety of subjects like wifi, GPS, 3G, keyboard, that were extracted with 60 person-hours of effort on Android bug data-set [18]. These word lists are not available for other 3 projects – Eclipse, Mozilla, and Open Office.
7) *LDA*: These word lists were extracted using topic modelling on all the bug reports from the Eclipse, Mozilla, and Open Office bug dataset by Alipour *et al.* 20 word lists were constructed by using LDA for all the three projects, and the process took around one person-hour for each project. The topics were unlabelled.
8) *Random English Words*: In order to determine that the specialized word lists were actually having a significant effect, random English word lists were used as well. These word lists are the same as those used by Alipour *et al.*, consisting of 26 lists each with 100 random words.

The first five lists, general software engineering, Android development, Eclipse documentation, Open Office documentation, and Mozilla documentation, were extracted by labelling chapters on the basis of the software-engineering processes like maintenance, testing. During labelling, similar chapters were grouped together under a single label. For example, chapters titled “Software testing techniques”, and “Software testing strategies”, were grouped under *software testing*. The last three lists were used by Alipour *et al.*. These word lists contain words that are more inclined to describe the software engineering processes while the ones used by Alipour *et al.* depict the

TABLE I. 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 II. 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

products, and specific problems in the software development in the bug reports.

To build the first five lists, the frequency of each word occurrences in the text under each label was recorded. Then, every word that appeared on a comprehensive list of stop words [19] was removed. Every remaining word that occurred less than 100 times was also removed for the textbooks. 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. However, no frequency cutoff was used for the extraction of features from documentation, as its more concise than the textbook. 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 III-D.

B. Bug-Report Preprocessing

The bug reports were pre-processed according to the methodology of Alipour *et al.*.

- Bugs lacking sufficient information to use in the deduplication process were discarded. This included bugs without Bug IDs as well as bugs marked as a duplicate where the corresponding duplicate Bug ID was not found in the repository.
- Stop words were removed from the description and title fields using a comprehensive list of English stop words.
- The reports were organized into "buckets", similar to the methodology of Sun *et al.* [3]. 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. This bucket would have introduced a strong bias in the results since such large clusters of duplicate bugs are uncommon in other bug repositories.

After this preprocessing step, three different subsets of the bug reports were constructed for the Android, Eclipse, Mozilla, and Open Office datasets, each containing a different ratio of duplicate to non-duplicate reports, in order to observe the effect of different ratios. The differing ratios were used to determine what effect the proportion of duplicate bug reports had on the results. The three subsets used included a set with 20% duplicates (as per Alipour *et al.*), 10% duplicates, and 30% duplicates. In each case, random selection without replacement from the original dataset was used, selecting as many reports as possible while maintaining the desired ratios. The focus, however, was on the 20% duplicate to 80% non-duplicate split used by Alipour *et al.*; no claim is made to the superiority of the 80/20 split over other ratios, rather, it is used to enable direct comparison of results. A comparison between results using software-literature context features with those achieved using LDA is also presented, along with results achieved using completely random word lists, as a sanity check.

C. Textual and Categorical Similarity Features

After the construction of bug report pairs, compute the textual, and categorical similarity of the bug pairs were computed. 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.* [3] and were also used in by Alipour *et al.* [4], [5].

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). Categorical fields (component and type) were compared using a simple binary rating resulting in a value of 1 if matching and 0 otherwise. This typically also includes a comparison for a product field, however this field was not specified in 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). Hence, a total of *seven* textual and categorical features were obtained.

The exact formulae used are below, where d_1 and d_2 are sample bug reports:

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

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

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

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

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

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

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

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. An example of the Textual and Categorical features table can be seen in Table III.

D. Contextual Features and Table Generation

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

TABLE III. 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 IV. EXAMPLE OF FINAL FEATURES TABLE

Bug pair		Features							
BugID$_1$	BugID$_2$	BM25F	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

BM25F scores for word lists: All contextual features were extracted using the BM25F algorithm to compare 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 III-A), where each word list was treated as text. This results in a new feature for each word list in that context. For example, the general software engineering context contained 13 word lists, so this resulted in 13 new features for each bug report. This procedure was repeated for each of the contextual categories under investigation : labelled LDA context, general software engineering context, Android development domain context, Documentation context, and random English words context. An example of the general software-engineering context features can be seen in Table II.

Feature table generation: Next, the contextual features for individual bug reports were calculated. Using these tables, a comparison feature table was constructed for pairs of bugs. Initially, pairwise comparisons were generated for the textual and categorical features as discussed in Section III-C. Then, the contextual features were added for each of the bug reports to these tables along with a cosine similarity feature based on the similarity between feature vectors of the two bugs. Given, n is the number of contextual features generated using n word lists for each bug report, cosine similarity of two reports is defined as:

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

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

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

TABLE V. ACCURACY AND KAPPA SCORES USING DIFFERENT CONTEXT FEATURES FOR THE 80-20 SPLIT FOR THE ANDROID BUG DATASET. ACCURACY SCORES ARE SHOWN WITH THEIR 95% CONFIDENCE INTERVAL. ALL THE KAPPA SCORES HAVE C.I. OF 0.01

(a) Android textbook

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	85.43±0.46%	0.509	61.07±0.72%	0.178
Logistic	88.28±0.41%	0.594	80.79±0.19%	0.090
SVM	87.02±0.43%	0.550	80.00±0.00%	0.000
C4.5	**92.12±0.43%**	**0.752**	**87.37±0.50%**	**0.583**

(b) Software-engineering textbook

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	85.85±0.50%	0.500	77.16±0.66%	0.040
Logistic	89.25±0.38%	0.634	80.93±0.17%	0.090
SVM	90.54±0.36%	0.680	80.00±0.00%	0.000
C4.5	**92.10±0.47%**	**0.752**	**88.29±0.48%**	**0.618**

(c) Android and Software-engineering textbook

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	85.61±0.46%	0.516	63.53±0.87%	0.205
Logistic	89.66±0.38%	0.650	81.43±0.27%	0.168
SVM	89.10±0.42%	0.630	80.00±0.00%	0.000
C4.5	**92.36±0.41%**	**0.756**	**87.52±0.48%**	**0.602**

(d) Labelled LDA features from Alipour *et al.* [4]

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	86.52±0.49%	0.583	81.46±0.50%	0.391
Logistic	92.41±0.36%	0.753	90.43±0.38%	0.672
SVM	92.40±0.35%	0.750	90.54±0.37%	0.680
C4.5	**93.62±0.60%**	**0.799**	**90.79±0.47%**	**0.711**

(e) Random words

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	51.94±0.47%	0.134	50.39±1.21%	0.117
Logistic	83.73±0.39%	0.385	80.56±0.30%	0.161
SVM	82.60±0.42%	0.362	80.56±0.18%	0.161
C4.5	**89.61±0.48%**	**0.667**	**85.32±0.48%**	**0.445**

E. Machine Learning and Evaluation Criteria

The contextual features extracted were meant to be utilized by machine-learning algorithms that could label a pair of bugs as “duplicate” or “non-duplicate”. Unlike in the work of Sun *et al.* [3] that queries only duplicate bugs, this approach considers true-negatives in the evaluation measure the accuracy of comparing two non-duplicates. This is especially important in scenarios where there is a lack of duplicate bug markup. The table containing all the features, textual, categorical and contextual, was provided as an input to Weka [20], 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. In order to avoid over fitting, 10-fold cross validation was used.

Evaluation of the performance of these models was done in terms of accuracy and Cohen's kappa coefficient. Accuracy is defined as the ratio of number of instances that are correctly classified 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 is defined in Equation (9), where $P(e)$ is the chance of correctly classifying an instance with the naïve classifier (i.e. predicting the majority instance in all cases). In this specific case, $P(e)$ is equal to the number of non-duplicates expressed as a percentage of the dataset.

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

0-R learner, Naïve Bayes, Logistic Regression, SVM (Weka's SMO) and C4.5 (Weka's J48) decision-tree classifiers were used with default parameters to evaluate the performance of software-literature context method features. The 0-R learner always chooses the majority class and is useful as it establishes a baseline performance for the classifiers.

IV. Results

How well do software-literature context features, extracted from documentation and textbooks, help answer the question, *"are these two bug reports duplicates or not?"* Using the set of bugs from each project repository, tables with a ratio of duplicates to non-duplicates of 20-80 were created. 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.[3]). The results are summarized in Table V, Table VI, Table VII, and Table VIII. To demonstrate the effect of using different ratios of duplicates to non-duplicates, tables with ratio of 30-70, and 10-90 were also created for all four projects, with results summarized in Table IX.

Android dataset : These results are largely comparable to those achieved by Alipour *et al.* using labelled LDA. The results for the *software-literature context method* using 80-20 split are shown in Table V. Table V (a) shows the results for the Android textbook features, while Table V (b) shows the results for the general software-engineering textbook features. There is a marginal difference between the two features, with Android textbook features performing better, possibly because they better capture the relevant context. Table V (c) depicts the results using both the general software engineering and Android textbook features, with a marginal improvement over the Android textbook features. Perhaps, the general software engineering chapters are already covered by the domain specific Android chapter, thus there is a limited gain. The results for the labelled LDA can be seen in Table V (d) while the random context results can be seen in Table V (e).

The best performing classifier, C4.5, classified 92.36% of the bug-report pairs correctly using both the Android textbook and general software-engineering textbook features, whereas labelled LDA correctly classified 93.62% of the reports. When the random-word list was used, only 89.61% of the reports were correctly classified. The software-engineering textbook and Android textbook features perform marginally worse with 92.1% and 92.12% accuracy respectively. Among the classifiers, C4.5 performs the best followed by SVM, Logistic Regression, Naïve Bayes and 0-R learner, in that order for the all features tables. Considering kappa scores, the classifiers using the *software-literature context method* features perform similarly to those using labelled LDA, especially when the results using random word lists are taken under consideration. For the best classifier, C4.5, the kappa score for the combined software engineering and Android textbook context is 0.756, while for the software engineering context is 0.752 and Android textbook context is 0.752. The kappa score for labelled LDA is 0.799, whereas the random words context yields a kappa score of 0.667.

For the classifiers using only contextual features, the accuracy and kappa scores were lower than the corresponding values for the classifiers with textual, categorical, and contextual features. The labelled LDA had the highest accuracy at 90.79% and kappa score of 0.711, followed by Software Engineering textbook context, combined Android textbook and Software Engineering textbook context, Android textbook context, and random context in that order.

While the labelled LDA used 72 contextual features generated from 72 word lists, this method used only 10 features for Android textbook and 13 features for general software engineering textbook. Although, it is always possible for good results to be due to over-fitting, the 10 features from the Android development context outperform the 13 features generated by the general software-engineering context, albeit marginally. Hence, we conclude that context is more important than the number of features.

Eclipse dataset : Results are as shown in Table VI. Table VI (a) depicts the results for Eclipse documentation features, while the Table VI (b) shows the results for general software-engineering textbook features. There is a small difference between the two, with Eclipse documentation performing better. Better performance from Eclipse documentation can be attributed to the more focused, and relevant context that Eclipse's own documentation provides. Table VI (c) shows the results using both, general software engineering textbook, and Eclipse documentation features. There is a marginal drop in the performance of the classifiers as compared to Eclipse documentation case. Table VI (d) and Table VI (e) depict the performance of the LDA features, and random context, respectively.

The LDA feature had the highest accuracy with the best performing classifier, C4.5, with 92.90% while Eclipse documentation is a close second with 92.86% accuracy. The general software-engineering textbook context performs marginally worse at 92.37% while the random context has 92.2% accuracy. The context combining the documentation and textbook gives marginally lower accuracy at 92.81% than the Eclipse documentation context. The kappa scores, for Eclipse documentation, and LDA have the highest value of 0.775 for the C4.5 classifier. The software-engineering textbook features perform marginally worse with a kappa score of 0.761. The context combining the two - documentation, and textbook context has a kappa score of 0.772, marginally lower than documentation context, but higher than the textbook context.

TABLE VI. ACCURACY AND KAPPA SCORES USING DIFFERENT CONTEXT FEATURES FOR THE 80-20 SPLIT FOR THE ECLIPSE BUG DATASET. ACCURACY SCORES ARE SHOWN WITH THEIR 95% CONFIDENCE INTERVAL. ALL THE KAPPA SCORES HAVE C.I. OF 0.01

(a) Eclipse Documentation

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	88.80±0.91%	0.65	61.28±1.30%	0.139
Logistic	91.01±0.74%	0.711	83.48±1.03%	0.329
SVM	91.08±0.88%	0.719	80.00±0.00%	0.000
C4.5	**92.86±0.76%**	**0.775**	**85.87±0.50%**	**0.521**

(b) Software-engineering textbook

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	88.28±0.91%	0.630	64.16±1.30%	0.150
Logistic	90.59±0.76%	0.698	80.61±0.98%	0.090
SVM	90.46±0.90%	0.704	80.00±0.00%	0
C4.5	**92.37±0.76%**	**0.761**	**84.508±1.09%**	**0.442**

(c) Eclipse documentation and Software-engineering textbook

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	88.63±0.86%	0.649	63.56±1.52%	0.179
Logistic	91.22±0.77%	0.718	84.27±1.31%	0.390
SVM	91.33±0.72%	0.726	80.00±0.00%	0.000
C4.5	**92.81±0.73%**	**0.772**	**85.87±0.99%**	**0.533**

(d) LDA features from Alipour *et al.* [5]

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	86.52±1.02%	0.583	74.37±1.32%	0.271
Logistic	91.31±0.73%	0.721	83.84±1.02%	0.383
SVM	91.37±0.86%	0.750	82.39±0.99%	0.232
C4.5	**92.90±0.73%**	**0.775**	**86.17±0.86%**	**0.514**

(e) Random words

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	82.69±1.14%	0.545	36.62±1.21%	0.048
Logistic	90.48±0.89%	0.693	83.16±0.95%	0.315
SVM	90.54±0.77%	0.703	80.67±1.24%	0.194
C4.5	**92.20±0.72%**	**0.755**	**83.17±0.98%**	**0.315**

TABLE VII. ACCURACY AND KAPPA SCORES USING DIFFERENT CONTEXT FEATURES FOR THE 80-20 SPLIT FOR THE OPEN OFFICE BUG DATASET. ACCURACY SCORES ARE SHOWN WITH THEIR 95% CONFIDENCE INTERVAL. ALL THE KAPPA SCORES HAVE C.I. OF 0.01

(a) Open Office documentation

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	82.65±1.01%	0.468	57.79±1.46%	0.123
Logistic	87.81±0.79%	0.587	80.89±0.99%	0.164
SVM	87.815±0.94%	0.590	80.00±0.00%	0.000
C4.5	**91.47±0.76%**	**0.723**	**84.23±1.02%**	**0.468**

(b) Software-engineering textbook

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	84.12±0.94%	0.436	65.01±1.44%	0.106
Logistic	87.01±0.80%	0.554	80.02±0.97%	0.005
SVM	86.93±0.95%	0.553	80.00±0.00%	0
C4.5	**90.73±0.78%**	**0.700**	**81.96±1.01%**	**0.342**

(c) Open Office documentation and Software-engineering textbook

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	76.31±1.09%	0.367	57.49±1.51%	0.130
Logistic	87.86±0.79%	0.589	81.74±0.94%	0.223
SVM	87.92±0.91%	0.595	80.00±0.00%	0.000
C4.5	**91.51±0.76%**	**0.728**	**84.35±1.01%**	**0.478**

(d) LDA features from Alipour *et al.* [5]

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	78.70±1.09%	0.275	71.95±1.24%	0.088
Logistic	86.73±0.82%	0.542	80.19±1.03%	0.057
SVM	86.50±0.97%	0.530	80.00±0.00%	0.000
C4.5	**90.71±0.77%**	**0.699**	**82.35±1.02%**	**0.357**

(e) Random words

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	41.22±1.69%	0.085	31.99±1.24%	0.029
Logistic	86.78±0.83%	0.545	80.20±1.03%	0.067
SVM	86.45±0.97%	0.527	80.67±0.18%	0.194
C4.5	**89.78±0.77%**	**0.669**	**81.95±1.02%**	**0.219**

Contextual features without the use of any categorical or textual features, performed similarly. LDA had the highest accuracy, marginally higher than the Eclipse documentation,and combined context of software textbook and documentation.

This method used between 13 features for general software-engineering textbook context and 19 features for Eclipse documentation context, while the LDA used 20 features and the random-word context used 26 features, with lowest accuracy and kappa scores of the all contexts. This result also constitutes evidence that context is more important than the number of features.

Open Office dataset : Results are shown in Table VII. Table VII (a) depicts the results for Open Office documentation features, while the Table VII (b) shows the results for the general software-engineering textbook features. There is a small difference between the two, with Documentation features performing better which can be credited to more relevant context that Open Office's own documentation provides. Table VII (c) shows the results using both, general software-engineering textbook, and Eclipse documentation features. There is a marginal drop in the performance of the classifiers as compared to Open Office documentation case. Table VII (d) and Table VII (e) depicts the performance of the LDA features, and random context, respectively.

The combined Office documentation and software-engineering features has the highest accuracy with the best performing classifier, C4.5, with 91.51% while open Office documentation is close second with 91.47% accuracy. The general software engineering textbook context performs marginally lower at 90.73% while the LDA is even lower at 90.71% accuracy. The random context has the lowest accuracy at 89.78%. The kappa scores, for the classifier with combined Open Office documentation and software engineering textbook features has highest score at 0.728 for the C4.5 classifier. The Open office features perform marginally worse with a score of 0.723 followed by software engineering textbook features with 0.7, LDA and random context with 0.699.

TABLE VIII. ACCURACY AND KAPPA SCORES USING DIFFERENT CONTEXT FEATURES FOR THE 80-20 SPLIT FOR THE MOZILLA BUG DATASET. ACCURACY SCORES ARE SHOWN WITH THEIR 95% CONFIDENCE INTERVAL. ALL THE KAPPA SCORES HAVE C.I. OF 0.01

(a) Mozilla documentation

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	82.65±0.63%	0.468	67.49±0.87%	0.237
Logistic	90.77±0.47%	0.698	82.42±0.64%	0.275
SVM	90.62±0.54%	0.692	80.00±0.00%	0.000
C4.5	**92.66±0.46%**	**0.765**	**84.23±0.68%**	**0.468**

(b) Software-engineering textbook

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	84.83±0.65%	0.557	61.65±0.99%	0.169
Logistic	90.45±0.47%	0.686	80.72±0.72%	0.089
SVM	90.41±0.53%	0.682	80.00±0.00%	0
C4.5	**92.66±0.47%**	**0.764**	**83.72±0.74%**	**0.426**

(c) Mozilla documentation and Software-engineering textbook

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	79.39±0.73%	0.462	57.49±1.04%	0.130
Logistic	90.80±0.42%	0.699	83.36±0.64%	0.345
SVM	90.73±0.54%	0.691	80.00±0.00%	0
C4.5	**92.89±0.43%**	**0.772**	**85.33±0.73%**	**0.511**

(d) LDA features from Alipour *et al.* [5]

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	82.44±0.71%	0.483	74.90±0.88%	0.262
Logistic	90.74±0.42%	0.695	83.62±0.65%	0.363
SVM	90.66±0.53%	0.684	80.23±0.53%	0.186
C4.5	**93.14±0.40%**	**0.780**	**86.02±0.40%**	**0.508**

(e) Random words

	Contextual, Categorical and Textual		Contextual Only	
Algorithm	Accuracy%	Kappa	Accuracy%	Kappa
0-R	80.00±0.00%	0.000	80.00±0.00%	0.000
Naive Bayes	63.28±1.29%	0.276	35.23±1.54%	0.041
Logistic	90.53±0.44%	0.687	80.99±0.77%	0.167
SVM	90.41±0.53%	0.672	80.00±0.00%	0
C4.5	**92.22±0.42%**	**0.755**	**83.92±0.66%**	**0.356**

Standalone contextual features without the use of any categorical or textual features, performed in a similar way as above, both in accuracy and kappa scores. The combined context of Open Office documentation and the software-engineering textbook had the highest accuracy, marginally higher than the documentation features, followed by LDA, software textbook, and random words.

As noted earlier, the general software textbook context used 13 features while Open Office documentation context used 22 features. LDA used 20 features while the random context uses 26 features, with lowest accuracy and kappa scores of the all contexts. The general software-engineering textbook context performs better than LDA despite using far lower number of features, demonstrating that context is more important than number of features.

Mozilla dataset : Results are shown in Table VIII. Table VIII (a) depicts the results for the Mozilla documentation features, while the Table VIII (b) shows the results for the general software-engineering textbook features. They both perform equally well. Table VIII (c) shows the results using both the general textbook and documentation features. There is an increase in the performance of the classifiers as compared to either the documentation or the general textbook cases. Table VIII (d) and Table VIII (e) depicts the performance of the LDA features and random context, respectively.

LDA features had the highest accuracy at 93.14% with C4.5 classifier while the combined context of documentation and textbook was close at 92.89%. The Mozilla documentation and software engineering textbook features performed equally well with 92.66% accuracy. Random context performed worst with 92.22% accuracy. LDA achieved the highest kappa score of 0.780, while the combined documentation-textbook context was close second, documentation context, general software engineering textbook context, and random context followed in that order.

Classifiers using only contextual features, without any categorical or textual features, performed in a similar way as above both in accuracy but somewhat differently in kappa scores. LDA had the highest accuracy, marginally higher than the combined context of Mozilla documentation, and software textbook. However, the combined context of Mozilla documentation, and software textbook had a marginally better kappa score than LDA. These are followed by documentation, software textbook, and random words in that order respectively.

Both general software engineering textbook context, and Mozilla documentation context used 13 features each. They had quite close accuracy to LDA that uses 20 features, while the random context using 26 features had the lowest accuracy and kappa scores.

Mozilla does not have a downloadable central repository of developer documentation, unlike the other projects considered, Eclipse, and Open Office. The Mozilla documentation is more like a wiki [17], available online, spread out amongst numerous short webpages. We could only label a few relevant parts of the documentation to extract the word list features. This could possibly explain the similar performance of documentation features versus textbook features. Whereas in the case of Eclipse, and Open Office, documentation features outperformed the general textbook features.

From the above discussion and analysis, it can be seen that for Android, labelled LDA performed better than the *software-literature context method*, which performed at par with unsupervised LDA for the rest of the projects, where the labelled LDA context was not available. The random context performed worst out of all the contexts considered, implying that the context matters more than the number of extra features introduced. The general software-engineering textbook features that were applied across all the four projects perform marginally worse than the project-specific software documentation, or software platform texts such as the Android development textbook. In the case of Android, and Mozilla, these general software engineering contexts performed at par with project-specific documentation contexts, while performing marginally lower on with the other projects. This indicates that the *software-literature context method* features extracted from literature that is closer to actual domain of a product result in better accuracy.

TABLE IX. RESULTS FROM USING CONTEXTS WITH DIFFERENT % OF DUPLICATES FOR C4.5

Android			**Eclipse**			**Mozilla**			**Open Office**		
Duplicates%	Accuracy	Kappa	Duplicates%	Accuracy	Kappa	Duplicates%	Accuracy	Kappa	Duplicates%	Accuracy	Kappa
10%	**95.09%**	0.718	10%	**95.47%**	0.735	10%	**96.18%**	0.772	10%	**95.49%**	0.717
20%	92.12%	0.752	20%	92.86%	0.775	20%	92.66%	0.765	20%	91.47%	0.723
30%	89.56%	**0.752**	30%	91.18%	**0.790**	30%	90.74%	**0.777**	30%	88.64%	**0.726**
(a) Android dev textbook			**(a) Eclipse Documentation**			**(a) Mozilla Documentation**			**(a) Open Office Documentation**		
Duplicates%	Accuracy	Kappa	Duplicates%	Accuracy	Kappa	Duplicates%	Accuracy	Kappa	Duplicates%	Accuracy	Kappa
10%	**95.36%**	0.733	10%	**95.28%**	0.723	10%	**96.06%**	0.763	10%	**95.21%**	0.700
20%	92.12%	0.752	20%	92.37%	0.761	20%	92.66%	0.764	20%	90.73%	0.700
30%	90.17%	**0.767**	30%	91.12%	**0.789**	30%	90.63%	**0.775**	30%	87.81%	**0.706**
(b) Software-engineering textbook			**(b) Software-engineering textbook**			**(b) Software-engineering textbook**			**(b) Software-engineering textbook**		
Duplicates%	Accuracy	Kappa	Duplicates%	Accuracy	Kappa	Duplicates%	Accuracy	Kappa	Duplicates%	Accuracy	Kappa
10%	**96.31%**	0.790	10%	**95.79%**	0.754	10%	**96.22%**	0.774	10%	**94.82%**	0.680
20%	93.62%	0.799	20%	92.90%	0.775	20%	93.14%	0.780	20%	90.71%	0.699
30%	91.56%	**0.799**	30%	90.71%	**0.699**	30%	91.33%	**0.792**	30%	87.37%	**0.699**
(c) Labelled LDA			**(c) LDA**			**(c) LDA**			**(c) LDA**		

Using different split ratios of duplicates to non-duplicates: When different splits were used to generate the dataset, there was a small difference in the resulting accuracy and kappa scores. The change in scores with respect to different splits can be seen in Table IX for four project datasets. However, it is interesting to note that as the percentage of duplicates increases, the accuracy increases while the kappa score decreases, which is natural given that kappa addresses the skewness of the data. In this paper, we focus primarily on the 80-20 split, as it gives a direct point of comparison with Alipour *et al.*'s work.

V. DISCUSSION

The best overall learner was C4.5 (Weka's J48) in terms of accuracy and Kappa score, across all bug-report repositories and features used. In general, the software-literature context method using a general software-engineering textbook and/or project-specific documentation fared at par with LDA approaches, but worse than labelled-LDA approaches. All of these approaches did better than the random-words approach.

In the case of the Android data set, the software-literature context features performed slightly worse than labelled-LDA 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. The *software-literature context method* features perform at par with LDA, only with marginal differences across the three datasets: marginally lower in the case of Mozilla, almost at parity in Eclipse, and marginally better in the case of Open Office. The time taken to extract the LDA word lists is around one person-hour while, as mentioned earlier, the software-literature context method required only half of a person-hour from a single author, per project. LDA requires extraction of all the bug descriptions, knowledge of using the sophisticated LDA tools, and tuning of parameters. Additionally, for both labelled, and unsupervised LDA the time and resources required may increase as the lists need to be updated when the number of bug reports in the bug-tracker system increases.

Software-literature features are simple, general and easy to extract and use. The process involves labelling chapters from software-literature sources and extracting word lists with simple tokenization. Some literature sources, such as general software-engineering textbooks, are relevant across all software projects and performed only marginally lower than the LDA, and other software project-specific features. For example, the context word list extracted through the labelled LDA process contained topics like "3G", which had not yet been coined when the 2001 edition of Pressman's book [7] was published; still the loss in performance was minor. The LDA context will had be to updated as and when new features are introduced into software projects, but not so with these general software-engineering features while also being applicable to all the software projects.

The performance of the generic software-engineering textbook features 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. Additionally, this concept can be applied in related software engineering tasks like concept location, or feature location.

> Labelled LDA is very time-intensive, whereas the software-literature context method results in huge time savings for the developers by many orders of magnitude. The *software-literature context method* for bug deduplication is at par with unsupervised LDA while reducing the time and improving generalizability.

VI. FUTURE WORK

There are many open questions in the field of contextual features used in software-engineering tasks such as duplicate bug-report detection.

(a) *What are good methods for representing contextual knowledge?* This work uses word lists, but can word distributions or n-gram corporae be used instead? Should TF-IDF be used to choose contextual words and what thresholds should be used? Should LDA be exploited to produce better contextual features?

(b) *How could multiple forms of documentation be exploited?* How can one extract good contexts from multiple textbooks or sources of documentation?

(c) *Does the domain really matter?* Given different domains of software, does the choice of domain have an effect on quality and value of contextual features? Do contexts from operating systems lead to better performance than those from video games?

(d) *How much do we gain from lower-level domain contexts?* Given a slice of a hierarchy of domains such as software engineering, databases, RDBMS Databses, MySQL concepts, what is the gain in terms of effort and effectiveness of contexts from each of these domains? How low level do the contexts have to be?

VII. Conclusions

This work demonstrates a method of improving the detection of duplicate bug reports using contextual information extracted from software-engineering textbooks and project documentation. Although, the accuracy was lower than that achieved by Alipour *et al.* in case of Android, it is important to note that the labelled LDA technique took 60 person-hours [6]. The *software-literature context method*, by comparison, took roughly a half of one person-hour resulting in only marginally lower accuracy and kappa scores than the labelled LDA.

In the case of unsupervised LDA, used on Eclipse, Mozilla, and Open Office the software-literature context method performed at par in Eclipse, marginally lower with Mozilla, and marginally higher with Open Office. This technique takes half the time of unsupervised LDA, without use of any sophisticated tools, optimization of parameters, or extraction of bug report descriptions, unlike LDA, while performing at parity.

Additionally, these general software-engineering features can be applied across a wide array of software projects, as demonstrated on four diverse datasets. The evaluation of features extracted from project documentation shows that developers can and should consider extracting their own contextual features from their own project documentation. Moreover, the LDA context, supervised or unsupervised, needs to be continually updated to incorporate new additions to the existing set of text from the bug reports. In comparison, the general software-engineering features extracted from a textbook published in 2001, performed only marginally worse, showing the robustness of the software-literature context method. The general software practice terms are used across all software platforms, hence software literature context features are applicable across all software platforms. Also, this method can find possible applications in other software-engineering tasks like feature location, or concept location. All the word lists and bug datasets used in this paper can be found online at: https://bitbucket.org/kaggarwal32/bug-deduping-dataset.

This paper proposes a simpler *software-literature context method* for timely detection of duplicate bug reports, reinforcing the previous findings of Alipour *et al.* that incorporating contextual data into software-engineering classification systems can lead to reductions in the manual effort required while maintaining or improving the accuracy of duplicate bug report detection systems.

References

[1] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007, pp. 499–510.

[2] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. 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*. ACM, 2010, pp. 45–54.

[3] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 253–262.

[4] A. Alipour, A. Hindle, and E. Stroulia, "A contextual approach towards more accurate duplicate bug report detection," in *Proceedings of the Tenth International Workshop on Mining Software Repositories*. IEEE Press, 2013, pp. 183–192.

[5] A. Alipour, "A contextual approach towards more accurate duplicate bug report detection," Master's thesis, University of Alberta, Fall 2013.

[6] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding android fragmentation with topic analysis of vendor-specific bugs," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 83–92.

[7] R. S. Pressman and W. S. Jawadekar, "Software engineering," *New York 1992*, 1987.

[8] M. L. Murphy, *The Busy Coder's Guide to Advanced Android Development*. CommonsWare, LLC, 2009.

[9] N. Klein, C. S. Corley, and N. A. Kraft, "New features for duplicate bug detection." in *MSR*, 2014, pp. 324–327.

[10] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful... really?" in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE, 2008, pp. 337–345.

[11] 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.

[12] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "Rebucket: A method for clustering duplicate crash reports based on call stack similarity," in *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 2012, pp. 1084–1093.

[13] A. Sureka and P. Jalote, "Detecting duplicate bug report using character n-gram-based features," in *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*. IEEE, 2010, pp. 366–374.

[14] A. Lazar, S. Ritchey, and B. Sharif, "Improving the accuracy of duplicate bug report detection using textual similarity measures," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 308–311.

[15] A. Kiezun. Basic tutorial eclipse 3.1. [Online]. Available: http://archive.eclipse.org/eclipse/downloads/drops/R-3.1-200506271435/org.eclipse.jdt.doc.user.3.1.pdf.zip

[16] Sun Microsystems. (2008) Openoffice.org 3.0 developer's guide. [Online]. Available: https://wiki.openoffice.org/w/images/3/34/DevelopersGuide_OOo3.0.0.odt

[17] Mozilla Developer Network and individual contributors. Mozilla developer guide. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide

[18] E. Shihab, Y. Kamei, and P. Bhattacharya, "Mining challenge 2012: The android platform," in *The 9th Working Conference on Mining Software Repositories*, 2012, p. to appear.

[19] C. Buckley and G. Salton. (2013, Dec.) Stop word list. [Online]. Available: http://www.lextek.com/manuals/onix/stopwords2.html

[20] G. Holmes, A. Donkin, and I. H. Witten, "Weka: A machine learning workbench," in *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*. IEEE, 1994, pp. 357–361.