A Contextual Approach towards More Accurate Duplicate Bug Report Detection

Anahita Alipour
Department of Computing Science
University of Alberta
Edmonton, Canada
alipour1@ualberta.ca

Abram Hindle
Department of Computing Science
University of Alberta
Edmonton, Canada
hindle1@ualberta.ca

Eleni Stroulia
Department of Computing Science
University of Alberta
Edmonton, Canada
stroulia@ualberta.ca

***Abstract*—Bug-tracking and issue-tracking systems tend to be populated with bugs, issues, or tickets written by a wide variety of bug reporters, with different levels of training and knowledge about the system being discussed. Many bug reporters lack the skills, vocabulary, knowledge, or time to efficiently search the issue tracker for similar issues. As a result, issue trackers are often full of duplicate issues and bugs, and bug triaging is time consuming and error prone.**

Many researchers have approached the bug-deduplication problem using off-the-shelf information-retrieval tools, such as BM25F used by Sun et al. In our work, we extend the state of the art by investigating how contextual information, relying on our prior knowledge of software quality, software architecture, and system-development (LDA) topics, can be exploited to improve bug-deduplication. We demonstrate the effectiveness of our contextual bug-deduplication method on the bug repository of the Android ecosystem. Based on this experience, we conclude that researchers should not ignore the context of software engineering when using IR tools for deduplication.

***Index Terms*—duplicate bug reports; triaging; textual similarity; contextual information; machine learning; information retrieval; deduplication**

I. Introduction

As new software systems are getting larger and more complex every day, software bugs are inevitable phenomenon. Bugs occur for a variety of reasons, ranging from ill-defined specifications, to carelessness, to a programmers misunderstanding of the problem, technical issues, non-functional qualities, corner cases, etc. [1], [2]. Recognizing bugs as a "fact of life", many software projects provide methods for users to report bugs, and to store these bug/issue reports in a bug-tracker (or issue-tracking) system. Addressing these bugs frequently accounts for the majority of effort spent in the maintenance phase of a software project's life-cycle. This is why, researchers have been trying to enhance the bug-tracking systems to facilitate the bug-fixing process [3], [4].

For several reasons, such as lack of motivation of users and defects in the search engine of the bug-tracking systems [4], the users of software systems may report some bugs that already exist in the bug-tracking system. These bug reports are called "duplicates". The word duplicate may also represent the bug reports referring to different bugs in the system that are caused by the same software defect.

Recently, researchers have paid notable attention to detection of duplicate bug reports. Without finding and marking duplicate bug reports, the triager[1] may triage/assign duplicate bug reports to different developers. In addition, when a bug report gets fixed, addressing the duplicates as independent defects is just a waste of time. Finally, identifying duplicate bug reports can also be helpful in fixing the bugs, since some of the bug reports may provide more useful descriptions than their duplicate [4]. Currently, detecting duplicate bug reports is usually done manually by the triager. When the number of daily reported bugs for a popular software is taken into consideration, manually triaging takes a considerable amount of time and the results are unlikely to be complete. In Eclipse, for example, two person-hours are daily being spent on bug triaging [5].

A number of studies have attempted to address this issue by automating bug-report deduplication. To that end, various bug-report similarity measurements have been proposed, concentrating primarily on the textual features of the bug reports, and utilizing natural-language processing (NLP) techniques to do textual comparison [6], [7]. Some of these studies also use categorical features extracted from the fields of bug reports (i.e. *component, version, priority, etc.*) [8], [9].

In this work, we introduce a new approach for improving the accuracy of detecting duplicate bug reports of a software system. Our approach exploits domain knowledge, about the software-engineering process in general and the system specifically, to improve bug-report deduplication. Essentially, rather than naively and exclusively applying information-retrieval (IR) tools, we propose to take advantage of our knowledge of the software process and product. Intuitively, we hypothesize that bug reports are likely to refer to software qualities, i.e., non-functional requirements (possibly being desired but not met), or software functionalities (linked to architectural components responsible for implementing them). Thus, we utilize a few software dictionaries and word lists, exploited by prior research, to extract the context implicit in each bug report. To that end, we compare the contextual word lists to the bug reports and we record the comparison results as

[1]The person who is in charge of processing the newly reported bugs and passing them to appropriate developers to get fixed.

new features for the bug reports, in addition to the primitive textual and categorical features of the bug reports, such as *description, component, type, priority, etc.* proposed in Sun *et al.*'s work [8]. Then, we utilize this extended set of bug-report features to compare bug reports and detect duplicates. Through our experiments, we demonstrate that the use of contextual features improves bug-deduplication performance. Also, we investigate the effect of the number of added features on bug-deduplication.

We evaluate our approach on a large bug-report data-set from the Android project, which is a Linux-based operating system with several sub-projects[2]. We examine 37236 Android bug reports. In this research, we are taking advantage of five different contextual word lists to study the effect of various software engineering contexts on the accuracy of duplicate bug-report detection. These word lists include: Android architectural words [10], software Non-Functional Requirements words [11], Android topic words extracted applying Latent Dirichlet Allocation (LDA) method [12], Android topic words extracted applying Labeled-LDA method [12], and random English dictionary words (as a control). To retrieve the duplicate bug reports, several well-known machine-learning algorithms are applied (using Weka [13]). To validate the retrieval approach we employed 10-fold cross validation. We indicate that our method results in 16.07% relative improvement in accuracy and an 87.59% relative improvement in Kappa measure (over the baseline).

This work makes the following contributions.

- We propose the use of domain knowledge about the software process and products to improve bug-deduplication performance.
- We demonstrate that our method improves the accuracy of duplicate bug-report detection by 16.07% and the Kappa measure by 87.59% (over the baseline).
- We systematically investigate the effect of considering different contextual features on the accuracy of bug-report deduplication.
- Finally, we posit a new evaluation methodology for bug-report deduplication, that improves the methodology of Sun et al.'s [8] by considering true-negative duplicate cases as well.

The paper is organized as follows. Section II presents some existing studies done on software bug reports as well as utilizing IR techniques in software engineering. In Section III, we discuss the software contextual data-sets used in our experiments. Section IV describes our duplicate bug-report detection methodology. In Section V, our case study of 37236 Android bug reports is explained to show the effectiveness of our approach on a large real data-set. At the end, Section VI presents the conclusion of this research.

II. Related Work

Information Retrieval (IR) is a popular topic in software engineering research due to the prevalence of natural language artifacts. In this section we cover relevant IR works and bug-deduplication studies.

A. IR in Software Engineering

Binkley *et al.* [14] applied a variety of IR techniques, including latent semantic indexing (LSI) — a generative probabilistic model for sets of discrete data proposed by Dumais *et al.* [15] — and Formal Concept Analysis (FCA) — a mathematical theory of data analysis using formal contexts and concept lattices explained by Ganter, B. et al. [16] — on different software repositories. They have addressed software problems like fault prediction, developer identification for a task, assisting engineers in understanding unfamiliar code, estimating the effort required to change a software system, and refactoring.

Marcus *et al.* have used LSI to map the concepts expressed by the programmers (in queries) to the relevant parts in the source code [17]. Their method is built upon finding semantic similarities between the queries and modules of the software.

Blei *et al.* [18] have described Latent Dirichlet Allocation (LDA), a generative model for documents in which each document is related to a group of topics. They have presented a convexity-based variational approach for inference and demonstrated that it is a fast algorithm with reasonable performance.

Hindle *et al.* have proposed and implemented a labeled topic extraction method based on labeling the extracted topics (from commit log repositories) using non-functional requirement concepts [19]. Their method is based on LDA topic extraction technique. They have selected the non-functional requirements concept as they believe these concepts apply across many software systems.

Poshyvanyk *et al.* have applied the FCA, LSI and LDA techniques in order to locate the concepts in the source code [20]. They have also used LDA to investigate conceptual coupling [21].

B. Bug Report Deduplication

Just *et al.* [3] believe that the current bug tracking systems have defects causing IR processes be less precise. By surveying 872 developers the authors conclude that issue trackers should improve their interfaces to augment the information they already provide developers.

In [7], Runeson *et al.* have developed a prototype tool to study the effect of NLP on detecting duplicate bug reports using the defect reports of Sony Ericson Mobile Communications. Their evaluations shows that about 66% of the duplicates can possibly be found using the NLP techniques. Also, they have studied different variants of NLP techniques like changing number of stop words to check for, and spell-checking and synonym replacement. But, these techniques could only make minor differences in results.

In [22], Nagwani *et al.* provide two different definitions for similar and duplicate bugs. Two bugs are similar when the same implementation behaviour is required for resolving these two bugs. Two bugs are duplicate when the same bug is reported by using different sentences in description. Then, the

[2]Android Operating System Project http://source.android.com/

bugs are compared utilizing some well-known string similarity algorithms and semantic similarity methods. Based on some specified thresholds, if all the similarity measures meet the thresholds, bug reports are duplicates; if some meet, the bug reports are similar.

Bettenburg *et al.* [4] propose an approach in which triaging is done by machine learners. Titles and descriptions (textual features of bug reports) are converted to word vectors. In this study, SVM and naive Bayes algorithms are utilized for automatic triaging. SVM demonstrates a better accuracy and the highest accuracy for SVM is 65%. Besides, they discuss the idea that duplicate bug reports can be useful as they provide more information about a software defect. So, the authors prefer to merge duplicate bug reports rather than removing them.

Jalbert *et al.* [9] have introduced a classifier for incoming bug reports which combines the categorical features of the reports, textual similarity metrics, and graph clustering algorithms to identify duplicates. In this method, bug reports are filtered based on an automatic approach. Their method is evaluated on a data-set of 29000 bugs from Mozilla Firefox. As a result, development cost was reduced by filtering out 8% of duplicate bug reports.

Wang *et al.* [23] used natural language information accompanied by execution information to detect duplicate bugs, evaluated on the Firefox and Eclipse bug repositories. Reports are divided into three groups: run-time errors, feature requests, and patch errors. They achieve better performance than relying solely on natural language information. This approach shows some promise behind using contextual information

Lotufo *et al.* [24] studied how a triager reads and navigates through a bug and made a bug summarizer using this research. They successfully evaluated the quality of their summarizer on a wide survey of developers.

Finally, Sun *et al.* [8] have proposed a new text-similarity-based duplicate bug-report retrieval model based on BM25F [6], a document similarity measurement method built upon $tf - idf$. In addition to textual features of bug reports, other categorical information from the bug reports, including *product, priority,* and *type* are utilized to retrieve duplicate bug reports. They evaluate their method by producing a list of candidate duplicate bug reports for every bug report marked as "duplicate" by the triager, to see if the correct duplicate is a candidate or not. The authors applied their technique on three software bug repositories from Mozilla, Eclipse, and OpenOffice and achieved duplicate bug-report detection improvement with 10-27% in recall rate@k ($1 \leq k \leq 20$) and 10-23% on average.

III. Data

The data-set used in this study involves Android bugs submitted from November 2007 to September 2012. After filtering unusable bug reports (the bug reports without necessary feature values such as Bug ID), the total number of bug reports is 37236 and 1063 of them are marked as duplicate. To study the effect of software-development contexts on detecting duplicate bug reports, we have taken advantage of several lists of contextual words as follows.

- **Android architecture words:** Guana *et al.* [10] produced a set of Android architecture words to categorize the Android bug reports based on architecture. Their word list is adopted from Android architecture documents and is organized in five word lists (one word list per Android architectural layer[3]) with the following labels: Applications, Framework, Libraries, Runtime, and Kernel.
- **Non-Functional Requirement (NFR) words:** Hindle and Ernst *et al.* [11] have proposed a method to automate labeled-topic extraction, built upon LDA, from commit-log comments in source control systems. They have labeled the topics from a generalizable cross-project taxonomy consisting of non-functional requirements such as portability, maintainability, efficiency, etc. They have created a list of software NFR words organized in six word lists with the following labels: *Efficiency, Functionality, Maintainability, Portability, Reliability*, and *Usability*.
- **Android topic words:** Han *et al.* [12] have applied both LDA and Labeled-LDA [25] topic analysis models to Android bug reports. We are using their Android HTC LDA topics, organized in 35 word-lists of Android topic words labeled as Topic*i* where i ranges from 0 to 34. We also use their Android HTC topics extracted by Labeled-LDA organized in 72 lists of words labeled as follows: *3G, alarm, android_market, app, audio, battery, Bluetooth, browser, calculator, calendar, calling, camera, car, compass, contact, CPU, date, dialing, display, download, email, facebook, flash, font, google_earth, google_latitude, google_map, google_navigation, google_translate, google_voice, GPS, gtalk, image, input, IPV6, keyboard, language, location, lock, memory, message, network, notification, picassa, proxy, radio, region, ringtone, rSAP, screen_shot, SD_card, search, setting, signal, SIM_card, synchronize, system, time, touchscreen, twitter, UI, upgrade, USB, video, voicedialing, voicemail, voice_call, voice_recognition, VPN, wifi,* and *youtube*.
- **Random English words:** To investigate the influence of contextual word lists on the accuracy of detecting duplicate bug reports, we have have created a collection of randomly selected English dictionary words. In other words, we have created this "artificial context" to study if adding noise data to the features of bug reports can improve deduplication even though the added data does not represent a valid context. This collection is organized in 26 word lists, labeled a through z. In each of these word lists there are 100 random English words starting with the same English letter as the label of the word list.

IV. Methodology

In the Android bug-tracking system[4], each bug report includes a *Bug ID, description, title, status, component, priority,*

[3]Android architecture words: http://source.android.com/tech/security/

[4]Android Issue Tracker: http://code.google.com/p/android/issues/list

type, version, open date, close date, and *Merge ID*. The status feature can have different values including "Duplicate" which means the bug report is recognized as a duplicate report by the triager [26]. To explain the functionality of Merge ID we bring the following example. Assume the bug report A is recognized as a duplicate of bug report B by the triager, the Merge ID feature of A refers to B's Bug ID. We call B the "immediate master" of A. Table I depicts some examples of duplicate bug reports with their immediate master reports in the Android bug-tracking system.

TABLE I: Examples of duplicate bug reports from Android bug-tracking system.

Pair	ID	Component	Priority	Type	Version	Status	Merge ID
1	13321	GfxMedia	Medium	Defect		New	
	13323	GfxMedia	Medium	Defect		Duplicate	13321
2	2282	Applications	Medium	Defect	1.5	Released	
	3462	Applications	Medium	Defect		Duplicate	2282
3	14516	Tools	Critical	Defect	4	Released	
	14518	Tools	Critical	Defect	4	Duplicate	14516

Table I shows examples of pairs of duplicate bug reports in Android and their categorical features. According to this table, these duplicate bug reports have similar categorical features. This motivates the use of categorical features in bug-deduplication.

In the rest of this section, we explain our methodology. Figure 1 displays the workflow of our method.

A. Preprocessing

After extracting the Android bug reports, we applied a preprocessing method consisting of the following steps:

1) First, we removed the bug reports without Bug IDs as well as the duplicate bug reports with immediate master reports not existing in the bug-tracking system.
2) Next, we removed the stop words from the textual features (*description* and *title*) of bug reports using a comprehensive list of English stop words[5].
3) We organized the bug reports in a list of buckets. A bucket is a data structure, introduced by Sun *et al.* [8], including one bug report as master and a list of bug reports as duplicates. All the bug reports are inserted in the same bucket with their immediate master bug report while the bug report with the earliest open time is the master report of the bucket.
   Then, we converted the bug reports into a collection of bug-report objects with the following properties: *Bug ID, description, title, status, component, priority, type, version, open date, close date,* and optional *master id*, which is the ID of the bug report which is the master report of the bucket including the current bug report.

B. Textual and Categorical Comparison

After preprocessing, we measure the pairwise similarity between every two bug reports based on their primitive features (*description, title, component, type, priority,* and *version*).

[5]Stop Words: http://www.link-assistant.com/seo-stop-words.html

Since the *title* and *description* of bug reports are textual features, a textual similarity measurement method is used to compare them between two bug reports. This measurement method is a customized version of BM25F for long queries proposed by Sun *et al.* [8]. Figure 2 describes the textual and categorical measurement formulas applied in our method. These formulas are adapted from Sun *et al.*'s paper [8].

$$comparison_1(d_1, d_2) = BM25F(d_1, d_2)//ofunigrams$$

$$comparison_2(d_1, d_2) = BM25F(d_1, d_2)//ofbigrams$$

$$comparison_3(d_1, d_2) = \begin{cases} 1 & \text{if } d_1.prod = d_2.prod \\ 0 & \text{otherwise} \end{cases}$$

$$comparison_4(d_1, d_2) = \begin{cases} 1 & \text{if } d_1.comp = d_2.comp \\ 0 & \text{otherwise} \end{cases}$$

$$comparison_5(d_1, d_2) = \begin{cases} 1 & \text{if } d_1.type = d_2.type \\ 0 & \text{otherwise} \end{cases}$$

$$comparison_6(d_1, d_2) = \frac{1}{1 + |d_1.prio - d_2.prio|}$$

$$comparison_7(d_1, d_2) = \frac{1}{1 + |d_1.vers - d_2.vers|}$$

Fig. 2: Categorical and textual measurements for comparison of a pair of bug reports [8].

The first comparison defined in Figure 2 is the textual similarity between two bug reports over the features *title* and *description*, computed by $BM25F$. The second comparison is similar to the first one, except that the features *title* and *description* are represented in bigrams (a bigram consists of two consecutive words). For more information about the implementation of BM25F please see the Sun *et al.*'s paper [8]. The remaining five comparisons are categorical comparisons.

Since the $comparison_3$ is comparing the *product* of bug reports, it is not applicable for our Android bug repository as the *product* feature of each Android bug report is not specified. So, we set the value of this feature to 0 for all the bug reports.

$Comparison_4$ compares the *component* features of the bug reports. The *component* of a bug report may specify an architecture layer or a more specific module within an architectural layer. The value of this measurement is 1 if the two bug reports belong to the same component and 0 otherwise.

$Comparison_5$ compares the *type* of two bug reports, i.e., whether they are both "defects" or "enhancements". This comparison has the value of 1 if the two bug reports being compared have the same *type* and 0 otherwise.

$Comparison_6$ and $comparison_7$ compare the *priority* and *version* of the bug reports. These measurements could have values between 0 and 1 (including 1).

The result of this step is a table including all the pairs of bug reports with the seven comparisons shown in Figure 2 and a column called class which reports if the two bug reports are in the same bucket or not. Table II shows a snapshot of this

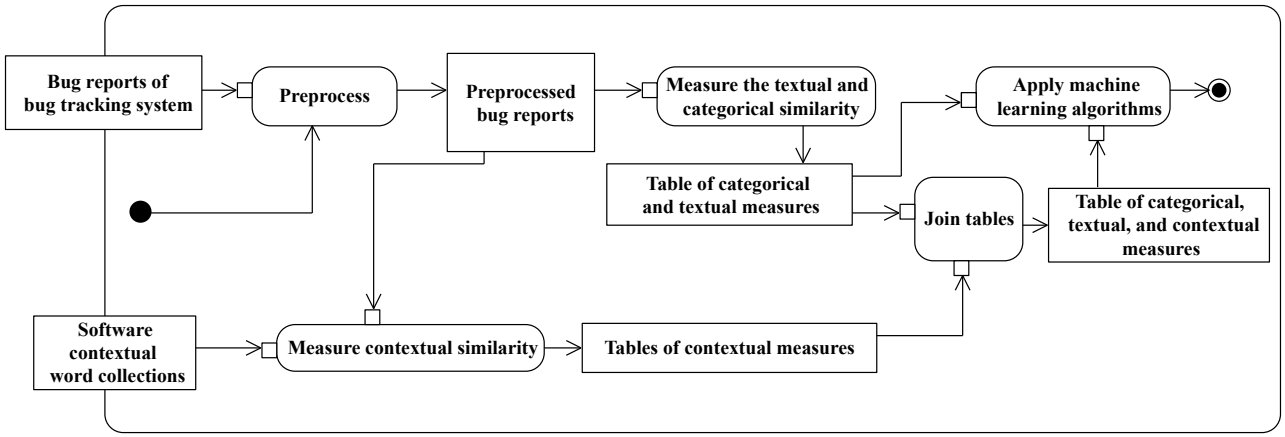

Fig. 1: Workflow of our methodology. The typical rectangles represent data and the rounded corner rectangles represent activities.

table with some examples of pairs of the Android bug reports. This table includes the measurements shown in Figure 2. The value of class column is "dup" if the bug reports are in the same bucket and "non" otherwise.

Regarding the number of bug reports in the Android bug-tracking system (37236), a huge number of pairs of bug reports are generated in this step (37236×37236). Consequently, we need to sample the records of the table before running the experiments. Since, there are only about 20000 of pairs of bug reports marked as "dup", and we want to create a set of bug report pairs including 20% "dup"s and 80% "non"s, we have selected 4000 "dup" and 16000 "non" pairs of reports randomly. So, we have 20000 sampled pairs of bug reports. The list of 20000 sampled bug-report pairs (with categorical and textual comparison measurements) is loaded into a SQL table called "textual_categorical". The schema of this table is shown in Table II.

C. Contextual Measurement

As mentioned in Section II, most of the previous research on detecting duplicate bug reports has focused on textual similarity measurements and IR techniques. Some approaches consider the categorical features of bug reports, in addition to the text. Here, we intend to describe our new approach which involves measuring the contextual similarity among the bug reports. We believe this new similarity measurement can help finding the duplicate bug reports more accurately by making the context of a bug report a feature during comparison.

In our method, we take advantage of the software contextual word collections described in Section III. We explain the contribution of context in detail, using the NFR context as an example. As mentioned in Section III, this contextual word collection includes six word lists (labeled as efficiency, functionality, maintainability, portability, reliability, and usability). We consider each of these word lists as a text, and calculate the similarity between each text and every bug report textually (using BM25F). For the case of NFR context, there are six BM25F comparisons for each bug report, which result in six new features for the bug reports. Table III shows the contextual features resulting from the application of the contextual measurements, using NFR context, for some of the Android bug reports. Each column shows the contextual similarity between the bug report and each of the NFR word lists. For example, the bug with the id 29374 seems to be more related to usability, reliability, and efficiency rather than the other NFR contexts.

The same measurement is done for the other contextual word collections as well. At the end, there will be five different contextual tables as we have five contextual word collections (Labeled-LDA, LDA, NFR, Android architecture, and English random words). These tables are inserted into five SQL tables called "Labeled_LDA_table", "LDA_table", "NFR_table", "architecture_table", "random_words_table". These are the tables called "tables of contextual measures" in Figure 1.

D. Calculating the Comparisons

In this phase of the process, we have the "textual_categorical" table for pairs of bug reports (as shown in Table II) and five tables reporting "contextual features" for individual bug reports, as described in Section IV-C. The next step involves a comparison of these contextual features for the pairs of bug reports.

As our research objective is to understand the impact that contextual analysis may have on bug-deduplication, in this phase, we aim to produce five different tables, each one including pairwise bug-report comparisons across (a) textual features, (b) categorical features and (c) one set of contextual features. One of these tables, the one corresponding to the "NFR" contextual feature is shown in Table V. As demonstrated in this view, the first seven columns are the same as the ones in Table II; they report the similarity measurements between the two bug reports according to textual and categorical features. Next are two families of six columns each, reporting the NFR contextual features for each of the two bug reports (with Bug $ID1$ and Bug $ID2$ respectively). The second to last column of the Table V reports the contextual similarity of the two bug reports based on these two column families. We consider the contextual features of the two bug reports as value vectors and measure the distance between these two

TABLE II: Some examples of pairs of bug reports with categorical and textual comparison values ("textual_categorical" table).

Bug ID1	Bug ID2	$BM25F_{uni}$	$BM25F_{bi}$	Product comp	Component comp	Type comp	Priority comp	Version comp	Class
3462	2282	1.5193	0	0	1	1	1	0.2857	dup
14518	14516	1.4841	0	0	1	1	1	1	dup
29374	3462	0.6282	0.1203	0	0	1	1	1	non
27904	14518	0.1190	0	0	0	1	0.3333	.1667	non

TABLE III: Examples of NFR contextual feature values for some Android bug reports

Bug ID	Efficiency	Functionality	Maintainability	Portability	Reliability	Usability
3462	3.4474	4.5729	1.3499	0.5653	1.5315	1.4094
2282	2.8829	2.5083	1.0662	3.3723	4.5321	4.9141
29374	3.8856	2.5235	0.1280	0.9888	3.2025	5.0744
27904	2.9330	1.0252	0.4990	0.0000	3.3571	4.5536

vectors using a cosine similarity measurement. The formula for calculating this similarity is shown below.

$$cosine\_sim = \frac{\sum_{i=1}^{n} C1_i \times C2_i}{\sqrt{\sum_{i=1}^{n}(C1_i)^2} \times \sqrt{\sum_{i=1}^{n}(C2_i)^2}}$$

In this formula, n is the number of word lists of the contextual data which is equal to the number of contextual features added to each bug report (in the case of NFR, n=6). $C1_i$ and $C2_i$ are the i^{th} contextual features added to first and second bug reports in the pair respectively. The cosine similarity feature is demonstrated in Table V. This table shows an example where bug reports with IDs 3462 and 2282, and bug reports with IDs 29374 and 3462 are compared in terms of NFR context. One of the records demonstrates the features for two bug reports belonging to the same bucket (with class value of "dup"). And, the other one shows a pair of bug reports existing in different buckets (with the class value of "non"). The Table V which includes textual, categorical, and the NFR contextual similarity measurements, is called the "NFR all-features_table". Note that there are five different such "all-features" tables, each one corresponding to a different context.

E. Machine Learning Evaluation

In this section, we discuss our use of machine learners on different parts of our data for deciding if a pair of bugs are duplicates or not. To retrieve the duplicate bug reports we are taking advantage of well-known machine learning classification algorithms. In each experiment, a table including pairs of bug reports with a particular combination of similarity metrics (i.e. textual, categorical, and contextual features) is passed to the machine learners. Each "all-features" table includes all the inputs necessary for our machine learners. The machine learner should decide about the class column's value for each pair of bug reports. In other words, given any pair of bug reports, the machine learner should decide if the pair is a "dup" (the bug reports in the pair are in the same bucket) or a "non" (the bug reports in the pair are not in the same bucket) based on some combination of the similarity columns of the table.

The classifiers we use are implemented by Weka [13]. The 0-R algorithm is utilized to establish the baseline. The other applied algorithms are C4.5, K-NN (K Nearest Neighbours), Logistic Regression, and Naive Bayes. K-NN tends to perform well with many features, but as well if K-NN works it implies that the input data has a fundamentally simple structure that is exploitable by distance metrics. We use the 10-fold cross validation technique to avoid over-fitting during training and evaluation.

The evaluation of the retrieval performance is measured by the following metrics: accuracy, kappa, and Area Under the Curve (AUC). Accuracy is the proportion of true results (truly classified "dup"s and "non"s) among all pairs being classified. The formula for accuracy is indicated bellow.

$$acc = \frac{|true\ dup| + |true\ non|}{|true\ dup| + |false\ dup| + |true\ non| + |false\ non|}$$

True and false "dup" are the pair bugs truly and wrongly recognized as "dup" respectively by machine learners. True and false "non" have the same definition but for the "non" class value.

Kappa measures the agreement between the machine learning classifier and the existing classes in the bug-tracking system. In other words, it is the agreement between the classifier and the triager of the Android bug-tracking system. The equation for kappa is:

$$kappa = \frac{Pr(a) - Pr(e)}{1 - Pr(e)}$$

$Pr(a)$ is the relative observed agreement between the machine learning classifier and the triager. The $Pr(e)$ is the hypothetical probability of chance agreement, using the observed data, to calculate the probabilities of each observer (machine learner or triager) randomly saying each category. If the classifiers are in complete agreement, then $kappa = 1$. If there is no agreement among the classifiers other than what would be expected by chance , then $kappa = 0$.

AUC is the area under the Receiver Operation Characteristic (ROC) curve. ROC curve is created by plotting he fraction of truly recognized "dup"s out of all recognized "dup"s (True Positive Rate) versus the fraction of wrongly recognized "dup"s out of all recognized "non"s (False Positive Rate) by the machine learners. AUC is the probability that a classifier will rank a randomly chosen "dup" instance higher than a randomly chosen "non" one (assuming that "dup" class has a higher rank than "non" class).

TABLE IV: Examples of predictions made by K-NN algorithm for the data-set including textual, categorical, and Labeled-LDA context's data

Pair	ID	Title	Component	Priority	Type	Version	Actual	Prediction
1	3063	Bluetooth does not work with Voice Dialer	Device	Medium	Defect		dup	dup
	8152	Need the ability to use voice dial over bluetooth.		Medium	Defect			
2	3029	support for Indian Regional Languages.....	Framework	Medium	Enhancement		dup	non
	4153	Indic fonts render without correctly reordering glyphs	GfxMedia	Medium	Defect			
3	8846	Bluetooth Phonebook Access Profile PBAP Character Problem		Medium	Defect	2.2	non	non
	4153	[ICS] Question of Google Maps' location pointer		Medium	Defect			
4	719	enhanced low-level Bluetooth support	Device	Medium	Enhancement		non	dup
	1416	Bluetooth DUN/PAN Tethering support	Device	Medium	Enhancement			

TABLE V: Examples of the records in the table containing categorical, textual, and contextual features for pairs of bug reports.

Bug ID1	Bug ID2	$comp_1$	...	$comp_7$	$Efficiency_1$	...	$Usability_1$	$Efficiency_2$	...	$Usability_2$	Cosine_sim	class
3462	2282	1.5193	...	0.2857	3.4474	...	1.4094	2.8829	...	4.9141	0.7279	dup
29374	3462	0.6282	...	1	3.8856	...	5.0744	3.4474	...	1.4094	0.7876	non

V. Case Studies

We applied our method on the Android bug-tracking system. To study the effect of contextual data on the accuracy of duplicate bug-report detection, we applied the classification algorithms on three different data-sets:

1) the data-set including all of the similarity measurements are shown in the "all-features" tables (such as Table V, the NFR all-features table);
2) the data-set including only the textual and categorical similarity measurements of the table; and
3) the data-set including only the contextual similarity measurement features.

As mentioned before, the table includes 20000 pairs of randomly selected bug reports with 20% "dup"s and 80% "non"s. All the experiments were carried out by Weka [13] application. The tables VI, VII, VIII, IX, X, and XI report the results of experiments.

A. The Effectiveness of Context

In this section we analyze the effect of context on detecting duplicate bug reports based on the results reported in the tables. The Table VI shows the statistical evaluation measurement values without considering the context of bug reports. This table demonstrates the resulting evaluation measures of the machine learners with the input data created by Sun *et al.*'s method [8]. Considering that the 0-R algorithm reports the baseline measurement values, their similarity measurement method could improve the accuracy by 4.52%, the kappa by 0.46, and the AUC by 0.28 (over the baseline).The maximum values are shown in bold in Table VI. This table demonstrates that Sun *et al.*'s method is definitely finding duplicates.

The tables VI, VII, VIII, IX, X, and XI report the statistical measurement results, using the contextual data-sets in bug-report similarity measurements. The highest value in each column is shown in bold. As reported in these tables, when the contextual data is used with textual and categorical measurements, the accuracy, kappa, and AUC are improved by 12.11%, 0.76, and 0.41 respectively (in comparison to the baseline). When considering the context only, the accuracy, kappa, and AUC are improved by 16.07%, 0.88, and 0.45 respectively over the baseline.The highest improvements are achieved utilizing the LDA and Labeled-LDA contextual data. This result is promising because LDA is an automatic method and not that expensive to run and if its topics can help boost deduplication performance then we have an automatic method of improving duplicate detection.

Table IV illustrates some examples of predictions made by K-NN machine learning algorithm for the data-set including textual, categorical, and Labeled-LDA context's data. The first pair of bug reports is correctly recognized as duplicates by the machine learner given that both of the reports are about "Bluetooth" which is an Android Labeled-LDA topic. For the same reason the pair 4 is recognized as a duplicate by the machine learner while the reports in this pair are not duplicates of each other. In pair 2, the bug reports are categorically different and also textually not similar in terms of Android Labeled-LDA topics, but they are wrongly classified as non-duplicates by the machine learner. In the pair 3, the reports are categorically similar and they are correctly recognized as non-duplicates as they are about two different Android Labeled-LDA topics.

Figure 3 shows the ROC curves for results of applying K-NN algorithm on the various "all-features" tables. It also displays the ROC curve for the "textual_categorical" table. The figure shows that the Labeled-LDA context outweighs the other ones. The "No context" curve shows the performance of K-NN algorithm using the data generated by Sun *et al.*'s measurements (only textual and categorical measurements) which show poor performance in comparison to the other curves. Thus the addition of extra features with or without Sun *et al.*'s features improves bug-deduplication performance.

Figure 4 presents the ROC curves for results of applying C4.5 algorithm on the "all-features" tables. It also indicates the performance of C4.5 on the "textual_categorical" table. This diagram shows a tangible gap between the performance of C4.5 using different contextual data-sets and its performance without using any context.

Taking into account the reported measurements, the contextual comparison results definitely outperform the results when using only the similarity measurements provided by Sun C. et

al. [8]. This implies that the context of bug reports includes useful information for bug-report deduplication. Furthermore this information is not evident in the textual measures. Also, we have evidence now that duplicate bug reports are contextually close to each other.

Duplicate bug reports have similar context.
Context provides different information than textual similarity.

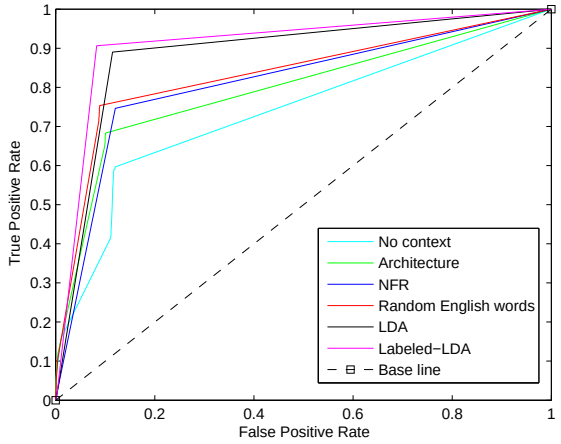

Fig. 3: ROC curve for K-NN algorithm.

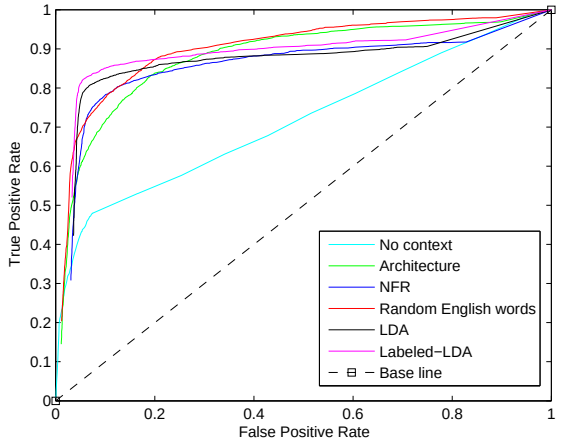

Fig. 4: ROC curve for C4.5 algorithm.

TABLE VI: Metrics for the experiments on the data-set including only textual and categorical comparisons, features used by Sun *et al.* [8]

Algorithm	Accuracy %	Kappa	AUC
0-R	80.0000%	0.0000	0.500
Logistic Regression	82.8300%	0.3216	0.814
Naive Bayes	78.6250%	-0.0081	**0.778**
C4.5	**84.5250%**	0.4324	0.716
K-NN	82.3800%	**0.4616**	0.737

B. *Effectiveness of Number of Features*

As mentioned before, each contextual data-set adds some new contextual features to each bug report. Number of these contextual features is equal to the number of word lists included in the contextual data-set. In this section, we analyze the influence of the number of added features (to the bug reports) on the bug-deduplication process.

Figure 5 shows the relationship between the kappa measure and the number of added features. Each box-plot in this figure represents the distribution of kappa values for each context reported by the machine learning classifiers (0-R, Naive Bayes, Logistic Regression, K-NN, and C4.5). In this diagram, there is a little difference between the performance of 26 Random English Word features and 6 NFR features, but NFR use 20 fewer features. Context is more important than feature count.

Moreover, we display the correlation between the number of added features and the AUC in Figure 6 by fitting a linear regression function (the slope of this line is 0.0012). The AUC measure for Naive Bayes, Logistic Regression, K-NN, and C4.5 is demonstrated in this figure. The measured correlation value for this figure is 0.46 which does not represent a high positive correlation.

Taking into account the above mentioned points, it is evident that adding more features can improve performance but contextually relevant features perform considerably better.

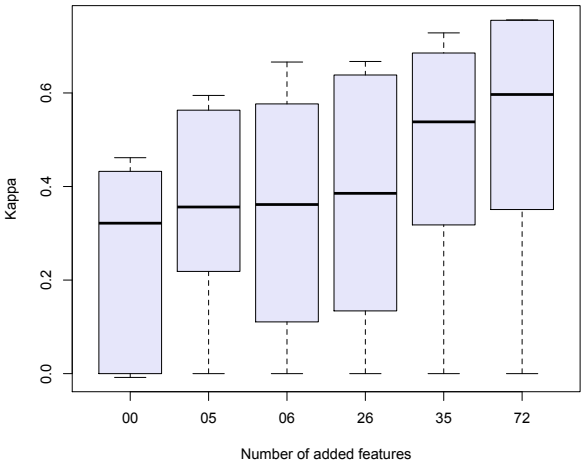

Fig. 5: Kappa versus number of added features. The x axis shows the number of the features each context adds to each bug report (which is equal to the number of word lists of the contextual data). The contexts from left to right are no context, architecture, NFR, Random words, LDA, and Labeled-LDA.

C. *Context Matters*

This paper describes one scenario where context matters. And, we also showed how software-development context matters in prior work [11]. This paper provides more evidence that we can gain in performance by including contextual features into our software engineering related IR tasks, whether it is bug duplication or LDA topic labelling and tagging. We hope this work serves as a call to arms for researchers to start building corpora of software concepts in order to improve automated and semi-automated software engineering tasks.

D. *Threats to Validity*

Construct validity is threatened by our word-lists in the sense of how they are constructed and if the word-lists actually

TABLE VII: Metrics for the experiments on the data-set including textual, categorical, and Android architectural context's data

Algorithm	Textual, Categorical, and Contextual			Contextual only		
	Accuracy %	Kappa	AUC	Accuracy %	Kappa	AUC
0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
Logistic Regression	83.060%	0.3562	0.829	79.965%	0.0005	0.618
Naive Bayes	77.950%	0.2185	0.732	75.255%	0.0825	0.603
C4.5	**87.990%**	**0.5947**	**0.880**	**91.690%**	**0.7083**	**0.916**
K-NN	85.580%	0.5632	0.794	86.330%	0.5553	0.843

TABLE VIII: Metrics for the experiments on the data-set including textual, categorical, and NFR context's data

Algorithm	Textual, Categorical, and Contextual			Contextual only		
	Accuracy %	Kappa	AUC	Accuracy %	Kappa	AUC
0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
Logistic Regression	83.325%	0.3615	0.833	79.995%	0.0014	0.617
Naive Bayes	78.735%	0.1106	0.758	77.880%	0.0509	0.619
C4.5	**89.450%**	**0.6661**	**0.856**	**96.145%**	**0.8792**	**0.952**
K-NN	85.295%	0.5766	0.813	83.165%	0.5222	0.788

TABLE IX: Metrics for the experiments on the data-set including textual, categorical, and random English words' data

Algorithm	Textual, Categorical, and Contextual			Contextual only		
	Accuracy %	Kappa	AUC	Accuracy %	Kappa	AUC
0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
Logistic Regression	83.730%	0.3854	0.844	80.200%	0.0543	0.661
Naive Bayes	51.845%	0.1341	0.665	39.260%	0.0515	0.606
C4.5	**89.995%**	**0.6673**	**0.901**	**91.590%**	**0.7101**	**0.917**
K-NN	87.955%	0.6384	0.834	87.620%	0.6119	0.863

TABLE X: Metrics for the experiments on the data-set including textual, categorical, and LDA context's data

Algorithm	Textual, Categorical, and Contextual			Contextual only		
	Accuracy %	Kappa	AUC	Accuracy %	Kappa	AUC
0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
Logistic Regression	86.780%	0.5382	0.886	80.590%	0.1447	0.732
Naive Bayes	77.290%	0.3179	0.767	73.565%	0.2523	0.712
C4.5	**91.245%**	**0.7284**	0.866	**96.070%**	**0.8759**	**0.946**
K-NN	88.615%	0.6854	**0.887**	89.345%	0.7034	0.894

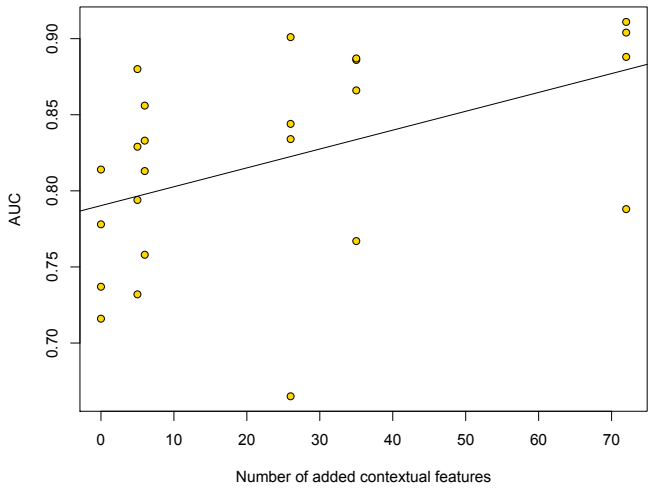

Fig. 6: AUC versus number of added features. The x axis shows the number of the features each context adds to each bug report. The contexts from left to right are no context, architecture, NFR, Random words, LDA, and Labeled-LDA.

represent context or just important tokens. Our measurements rely on the status of bug reports in the Android bug-tracking system that has a huge number of bug reports not processed by the triager (have the status value of "New"). And, there may be many duplicate bug reports among them. Also, the Android bug-tracking system includes only 1063 bug reports labeled as "duplicate" out of 37236 bug reports. There are likely many unlabeled duplicate bug reports.

We address internal validity by replicating past work (Sun *et al.*) but also by evaluating both on true negatives (non-duplicates) and true positives (duplicates), where as Sun *et al.*'s methodology only tested for recommendations on true positives. Furthermore internal validity is bolstered by searching for rival explanations of increased performance by investigating the effect of extra features.

External validity is limited by the sole use of the Android bug tracker, but this is a very large project (an OS and applications) so the breadth of the Android sub-projects provides some form of generality.

VI. Conclusion

In this paper, we have exploited the domain knowledge and context of software development to find duplicate bug reports. By improving bug deduplication performance companies can

TABLE XI: Metrics for the experiments on the data-set including textual, categorical, and Labeled-LDA context's data

Algorithm	Textual, Categorical, and Contextual			Contextual only		
	Accuracy %	Kappa	AUC	Accuracy %	Kappa	AUC
0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
LogisticRegression	88.125%	0.5967	0.904	82.605%	0.3151	0.798
Naive Bayes	79.655%	0.3508	0.788	77.560%	0.3082	0.747
C4.5	**92.105%**	0.7553	0.888	**95.430%**	**0.8574**	**0.939**
K-NN	91.500%	**0.7561**	**0.911**	92.405%	0.7801	0.921

save money and effort spent on bug triage and duplicate bug finding. We use contextual word lists to address the ambiguity of synonymous software-related words within bug reports written by users, who have different vocabularies. We replicated Sun *et al.*'s [6] method of textual and categorical comparison and extended it by adding our contextual similarity measurement approach. We have utilized the contexts of Android architecture, non-functional requirements (NFRs), and the Android LDA-extracted topics (extracted by LDA and Labeled-LDA). By including the overlap of context as features we found that our contextual approach improves the accuracy of bug-report deduplication by 11.55% over Sun *et al.*'s [8] method. This implies that by addressing the context of software engineering and relying on prior knowledge of software development we can boost bug de-duplication performance. We conclude that to improve duplicate bug-report detection performance one should consider, and not ignore, the domain and context of software engineering and software development.

ACKNOWLEDGEMENTS

This work is partially supported by Natural Sciences and Engineering Research Council (NSERC), Alberta Innovates Technology Futures (AITF), and International Business Machines (IBM) corporation.

REFERENCES

[1] T. Nakashima, M. Oyama, H. Hisada, and N. Ishii, "Analysis of software bug causes and its prevention," *Information and Software Technology*, vol. 41, no. 15, pp. 1059–1068, 1999.

[2] S. Hangal and M. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th international conference on Software engineering*. ACM, 2002, pp. 291–301.

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

[4] 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.

[5] J. Anvik, L. Hiew, and G. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 361–370.

[6] C. Sun, D. Lo, X. Wang, J. Jiang, and S. 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.

[7] 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.

[8] C. Sun, D. Lo, S. 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.

[9] 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.

[10] V. Guana, F. Rocha, A. Hindle, and E. Stroulia, "Do the stars align? multidimensional analysis of android's layered architecture," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE, 2012, pp. 124–127.

[11] A. Hindle, N. Ernst, M. Godfrey, and J. Mylopoulos, "Automated topic naming to support cross-project analysis of software maintenance activities," in *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 2011, pp. 163–172.

[12] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding android fragmentation with topic analysis of vendor-specific bugs."

[13] G. Holmes, A. Donkin, and I. 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.

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

[15] S. Dumais, G. Furnas, T. Landauer, S. Deerwester, S. Deerwester *et al.*, "Latent semantic indexing," in *Proceedings of the Text Retrieval Conference*, 1995.

[16] B. Ganter, R. Wille, and R. Wille, *Formal concept analysis*. Springer Berlin, 1999.

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

[18] D. Blei, A. Ng, and M. Jordan, "Latent dirichlet allocation," *the Journal of machine Learning research*, vol. 3, pp. 993–1022, 2003.

[19] A. Hindle, N. Ernst, M. W. Godfrey, R. C. Holt, and J. Mylopoulos, "Whats in a name? on the automated topic naming of software maintenance activities," *submissio n: http://softwareprocess. es/whats-in-a-name*, vol. 125, pp. 150–155, 2010.

[20] D. Poshyvanyk and A. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," in *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*. IEEE, 2007, pp. 37–48.

[21] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical Software Engineering*, vol. 14, no. 1, pp. 5–32, 2009.

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

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

[24] R. Lotufo, Z. Malik, and K. Czarnecki, "Modelling the hurriedbug report reading process to summarize bug reports."

[25] D. Ramage, D. Hall, R. Nallapati, and C. Manning, "Labeled lda: A supervised topic model for credit attribution in multi-labeled corpora," in *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1-Volume 1*. Association for Computational Linguistics, 2009, pp. 248–256.

[26] (2013) Life of a bug. [Online]. Available: http://source.android.com/source/life-of-a-bug.html