**Noname manuscript No.**
(will be inserted by the editor)

# A Contextual Approach towards More Accurate Duplicate Bug Report Detection and Ranking

**Abram Hindle · Anahita Alipour · Eleni Stroulia**

Received: date / Accepted: date

**Abstract** The issue-tracking systems used by software projects contain issues, bugs, or tickets written by a wide variety of bug reporters, with different levels of training and knowledge about the system under development. Typically, reporters lack the skills and/or time to search the issue-tracking system for similar issues already reported. As a result, many reports end up referring to the same issue, which effectively makes the bug-report triaging process time consuming and error prone.

Many researchers have approached the bug-deduplication problem using off-the-shelf information-retrieval (IR) tools. In this work, we extend the state of the art by investigating how contextual information about software-quality attributes, software-architecture terms, and system-development topics can be exploited to improve bug deduplication. We demonstrate the effectiveness of our contextual bug-deduplication method at ranking duplicates on the bug repositories of the Android, Eclipse, Mozilla, and OpenOffice software systems. Based on this experience, we conclude that taking into account domain-specific context can improve IR methods for bug deduplication.

**Keywords** Issue-tracking systems · Bug-tracing systems · Duplicate bug reports · Triaging · Bug deduplication · Information retrieval · Software context

A. Hindle
Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada
E-mail: abram.hindle@ualberta.ca

A. Alipour
Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada
E-mail: alipour1@ualberta.ca

E. Stroulia
Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada
E-mail: stroulia@ualberta.ca

## 1 Introduction

As new software systems are getting larger and more complex every day, software bugs are an inevitable phenomenon. Software development is an evolutionary process where, after the first release, users and testers submit bug reports that will drive the next software release. Bugs arise during different phases of software development, from inception to deployment. They occur for a variety of reasons, ranging from ill-defined specifications, to developers' misunderstanding of the problem, to careless coding practices, to the complexity of technical issues, etc. [14, 24].

Recognizing bugs as a "fact of life", many software projects provide services for users to report bugs, and to store these reports in a bug-tracker (or issue-tracking) system. Bug-tracking systems, such as Bugzilla [29] and Google's issue tracker [5], enable users and testers to report their findings in a central repository with a short description and a longer summary, akin to a message subject line and body, as well as tool-specific additional information. The system uses this information to categorize and, possibly, further annotate the bug report. This enables developers to query for bug reports based on a combination of textual and categorical (attribute-based) queries.

Bug reporting is an uncoordinated process, with multiple system stakeholders reporting issues as they experience them. As a result, an incoming report may refer to a bug that has already been reported in the bug-tracking system. These bug reports are called "duplicates". Researchers have posited several reasons for duplicate bug reports [6], including user inexperience, poor search functionality of bug trackers, and intentional/accidental re-submissions of the same bug report. It is therefore necessary to inspect new bug reports and decide whether they refer to bugs that have already been reported. If the incoming report is about a new bug, then it should be assigned to an developer with the knowledge and skills necessary to fix it. If, on the other hand, the bug report is a duplicate, it should simply be attached to the original "master" report to provide additional information to the developer assigned to it. This process is referred to as *triaging*. Triaging has important economic implications. As duplicate bugs are associated to a master bug, they are examined together and fixed together, thus avoiding duplication of the developers' debugging effort [6].

Duplicate-report detection and bug triaging have been mostly manual procedures carried out by the triagers. Considering the typical number of bugs reported daily for popular software systems, manual triaging requires a significant amount of effort and time. For instance, Mozilla reported in 2005 that "everyday, almost 300 bugs appear that need triaging" [3], two person-hours are daily being spent on bug triaging in Eclipse [4]. Even then, the results of the triaging process are unlikely to be completely accurate. This problem has recently motivated a line of bug-deduplication research that has explored multiple variants of information-retrieval methods, which, given a bug description, recognize similar bugs in the bug-tracking system and order them in decreasing rank of similarity. The intent is to focus the attention of the triager on a

few duplicate candidates in order to reduce the triaging effort while, at the same, improving the de-duplication accuracy.

The work described in this paper advances the state-of-the-art in bug-deduplication by demonstrating how the accuracy of the process can be improved by taking into account the "software-system context" in analyzing the bug reports. Intuitively, the thesis of our work is that bug reports should not be viewed as "documents" to be compared against each other with out-of-the-box IR methods; instead, any similarity analysis should be aware of the fact that domain-specific terms in the bug reports are essential in assessing the likelihood that two reports are duplicates. Bug reports are likely to refer to software qualities when, for example, end users experience poor performance below their expectations, including phrases such as "slow system response". They may also refer to software functionalities associated with the architectural components responsible for implementing them, when testers identify undesired behaviors with phrases such as "notification system failure". By making deduplication techniques aware of these contexts, we can improve their performance. To that end, we have developed and utilized several software vocabularies representing functional and non-functional requirements contexts, to extract the context implicit in each bug report, which can then be used as an additional feature of the bug report.

To evaluate the usefulness of our approach we experiment with four large bug repositories from the Android, Eclipse, Mozilla, and OpenOffice projects, taking advantage of five different contextual word lists including architectural words [12], software non-functional requirements words [16], topic words extracted applying the Latent Dirichlet Allocation (LDA) method [13], topic words extracted by the Labeled-LDA method [13], and random English dictionary words (as a control). We comparatively analyzed the degree to which these contexts could improve the deduplication accuracy of several well-known machine-learning classifiers. Our results demonstrate that our method offers up to 11.5% and 41/% relative improvements in accuracy and Kappa measures respectively, over the current state-of-the-art as exemplified by the work of Sun *et al.* [30]). From a methodological perspective, our work argues that the Mean Average Precision (MAP) measure is a very useful quality indicator for bug-deduplication methods. Since triagers are less likely to examine items later in a ranked list, any arbitrary threshold for how many duplicate candidates to present to triagers is difficult to rationalize. Therefore, the MAP measure that captures average precision at different thresholds is a better indicator than measuring precision-at-a-threshold.

This work makes the following contributions:

1. We propose the use of domain knowledge about the software development process and products to improve the bug-deduplication performance. We systematically investigate the effect of considering different contextual features on the accuracy of bug-report deduplication.
2. We posit a new evaluation methodology for bug-report deduplication (by applying machine learning classifiers), that improves the methodology of

Sun et al. [30] by considering true-negative duplicate cases as well true-positive ones.
3. We demonstrate that our contextual approach is able to improve the accuracy of duplicate bug-report detection by up to 11.5%, the Kappa measure by up to 41%, and the AUC measure by up to 16.8% over the Sun *et al.*'s method [30].
4. We improve the quality of the list of candidate duplicates and consequently the MAP measure by 7.8-9.5% over Sun *et al.*'s approach [30].
5. We describe a concrete disk-friendly indexing approach to answer multiple deduplication queries quickly called FastREP.

This paper extends our Mining Software Repositories 2013 paper [2] in two important ways. First, we evaluate our approach in the context of the candidate-duplicate ranking, as opposed to the duplicate-or-not decision. Second, in order to be able to do a more direct comparison with the work of Sun *et al.* [30], we apply our method to the same repositories, in addition to our original Android repository.

The rest of this paper is organized as follows. Section 2 presents an overview of the related work. Section 3 provides detailed information about the datasets exploited in our experiments. In Section 4 we discuss our approach for detecting duplicate bug reports. In Section 5, we report the results of our experiments on four different real world bug repositories including Android, Eclipse, Mozilla, and OpenOffice bug reports. Finally, we conclude in Section 6, summarizing the substantial points and contributions made in this work and propose some potential future work.

## 2 Related Work

Several researchers have studied duplicate bug-report detection. A number of these approaches exclusively exploit the IR techniques and the textual features of bug reports to identify duplicate bug reports. Some convert the textual features of the bug reports to word count vectors and compare them using textual comparison functions like the cosine similarity metric [15, 23, 28]. Yet others consider the difference in time, milestone or version [28, 30].

Sureka *et al.* [32] proposed a method that relies exclusively on the textual features of the bug reports to recognize duplicates. The main novelty of this approach is exploiting the character-level representations versus word-level ones, to gain robustness in the face of typos commonly found in bug-reports. The overall similarity score between the reports is calculated based on the following parameters: number of shared character n-grams between the two bug reports; and number of the character n-grams extracted from the title of one bug report present in the description of the other one. Applied against some Mozilla and Eclipse bug reports, this technique achieved up to 0.34 recall.

Sun *et al.* [31] proposed a novel text-based similarity measurement method in which the duplicate bug reports are organized in *buckets*, where every *bucket* includes a single master bug report and zero or more duplicate bug reports.

In this method, a Support Vector Machine (SVM) is exploited to predict the duplicate reports based on their textual features.

Some approaches take into account the stack-trace and execution information when comparing the bug reports. Wang *et al.* [36] reported a study in which two both are calculated for a pair of bug reports: the first is based on natural-language content, and the second is based on execution-information content. This method was able to detect 67%-93% of duplicate bug reports within the Firefox bug repository.

A third style of duplicate-detection approaches involves similarity assessment of the bug-reports categorical (non-textual) features. Jalbert *et al.* [18] proposed a duplicate-or-not classifier that combines the categorical features of the bug reports (features such as severity, operating system, and number of associated patches), textual similarity measurements, and graph clustering algorithms to identify duplicate bug reports. They evaluated against the Mozilla bug repository and were able to detect and filter 8% of duplicate bug reports automatically.

In this paper, we replicate the work of Sun *et al.* [30] that uses both textual and categorical features (including product, component, type, priority, and version) to compare bug reports. They proposed using a textual similarity metric called BM25F, to compare long queries such as bug reports descriptions. Moreover, they developed seven comparison metrics illustrated in Figure 2 to compare two bug reports in terms of their textual and categorical characteristics. To combine these comparisons, they proposed a linear function called REP:

$$REP(d, q) = \sum_{i=1}^{7} \omega_i \times comparison_i \tag{1}$$

Where $d$ and $q$ are two bug reports being compared; $comparison_i$s are the comparisons indicated in Figure 2; and $\omega_i$ are the weights for each comparison. In this method, every single incoming duplicate bug report is compared against all the existing buckets using the REP function in order to produce a sorted list of candidate masters. For evaluation the authors utilized two measures: recall and Mean Reciprocal Rank (MRR). The MRR measure is the average of the reciprocal ranks of the results for a sample of queries, where the reciprocal rank of a query response is the multiplicative inverse of the rank of the first correct answer. They [30] reported 10-27% improvement in recall rate@k ($1 \leq K \leq 20$) and 17-23% in MRR over the state-of-the-art.

More recently, Nguyen *et al.* [25] proposed a novel technique called *Duplicate Bug report Topic Model* (DBTM) that uses topic models to detect duplicate bug reports. They employ a LDA-based technique called T-Model to extract the topics from the bug reports. To measure the textual similarity between the bug reports, they use the BM25F method [30] and apply Ensemble Averaging, a machine-learning technique, to combine topic-based and textual metrics. This approach provides a list of top-k similar bug reports for every new report. The authors performed their experiments on OpenOffice, Eclipse,

and Mozilla bug repositories and reported up to a 20% relative improvement in top-k accuracy over their prior work on REP [30]. Top-k accuracy is the percentage of queries where the duplicate report appears in the top-k results. They did not compare against REP in terms of MRR or MAP, but trained DBTM using a measure equal to MRR.

### 2.1 Contextual Bug Report Deduplication

In this paper, we extend our recent work published in (Mining Software Repositories) MSR 2013. In our MSR paper [2], we developed a method to identify duplicate bug reports based on their contextual features, in addition to their textual and categorical fields. To implement this method, we exploited software contextual vocabularies, each consisting of a set of contextual word lists about software architectural words, software non-functional requirement words, topic words extracted by LDA, topic words extracted by Labeled-LDA, and random English words (as a control). Given these contextual words, we proposed several new features for the bug reports by comparing each contextual word list to the textual features of the bug reports (description and title) using the BM25F metric employed by Sun *et al.* [30].

To compare the bug reports in terms of both their textual and categorical features, we applied Sun *et al.*'s [30] comparison metrics illustrated in Figure 2. We then created a data-set including pairs of bug reports, including their textual, categorical, and contextual features, and provided this data-set to a number of machine-learning classifiers (using the 10-fold cross-validation experiment design) to decide whether the two bug reports in each record are duplicates or not. We conducted our experiments on bug reports from Android bug repository and succeeded in improving the accuracy of duplicate bug-report identification by 11.5% over the approach of Sun *et al.* [30]. We also investigated the influence of the number of added features on accuracy of the bug report deduplication by applying the random English words context which resulted in a poor performance. These results led us to the conclusion that, indeed, it is context that improves the deduplication performance, and not simply the number of added features to the bug reports.

In this study, we extended this prior work [2] by evaluating our method more broadly, against the Eclipse, Mozilla, and OpenOffice bug repositories, in addition to the Android bug repository we used in our original study. For these new repositories, our results show an improvement to the deduplication accuracy by up to 0.7% in accuracy, 2% in Kappa and 0.5% in AUC, which is not as significant as the improvement achieved for Android repository. In addition, we extended our duplicate retrieval method with three different bug-report similarity criteria, i.e. *i.e.,* cosine similarity, Euclidean distance, and logistic regression. As a result, for every incoming bug report, a sorted list of candidate duplicates (based on a specific similarity criterion) is provided to the triager to make the final decision about the actual duplicates of the

incoming report. We evaluate our bug-report retrieval method using the Mean Average Precision (MAP) metric.

## 3 The Data Set

As we mentioned earlier, in this study we examine four large bug repositories are: Android, Eclipse, Mozilla, and OpenOffice. Android is a Linux-based operating system with several sub-projects. The Eclipse, Mozilla, and OpenOffice bug repositories are adapted from Sun *et al.*'s paper [30]. Eclipse is a popular open source integrated development environment for Java development primarily (although more languages are also supported). OpenOffice is a well-known open source office suite, with several sub-projects including a word processor (Writer), a spreadsheet (Calc), a presentation application (Impress), a drawing application (Draw), a formula editor (Math), and a database management application. Mozilla is a company that develops free software, best known for producing the Firefox web browser, Thunderbird, Firefox Mobile, and Bugzilla.

**Table 1** Size of Repositories

| Dataset | #Bugs | #Duplicates | Period | | #Duplicate Including Buckets |
|---|---|---|---|---|---|
| | | | From | To | |
| Android | 37536 | 1361 | 2007-11 | 2012-09 | 737 |
| Eclipse | 43729 | 2834 | 2008-01 | 2008-12 | 2045 |
| Mozilla | 71292 | 6049 | 2010-01 | 2010-12 | 3790 |
| OpenOffice | 29455 | 2779 | 2008-01 | 2010-12 | 1642 |

Table 1 reports some interesting properties of each bug repository we studied. The last column in the table reports the number of bug buckets in each repository. As described in section 2, a bucket is a data structure proposed by Sun *et al.* [30] in which all the reports that are duplicates of each other are stored in one bucket and the one submitted earlier than others is called the "master" report. Also, Figure 1 illustrates the distribution of duplicate bug reports in the buckets for Android, Eclipse, Mozilla, and OpenOffice repositories.

Although each bug repository has distinct categorical and textual fields, many of these fields are analogous or shared with most bug-tracker repositories (e.g., description, severity, milestone). The fields of interest that we consider in our experiments are described in Table 2.

As indicated in Table 2, in our study the bug reports include the following fields: *description, summary, status, component, priority, type, version, product* and *Merge_ID*. The status feature can have different category values including "Duplicate", which means the bug report is recognized as a duplicate report by a triager or another user. For example, assume that bug report A is

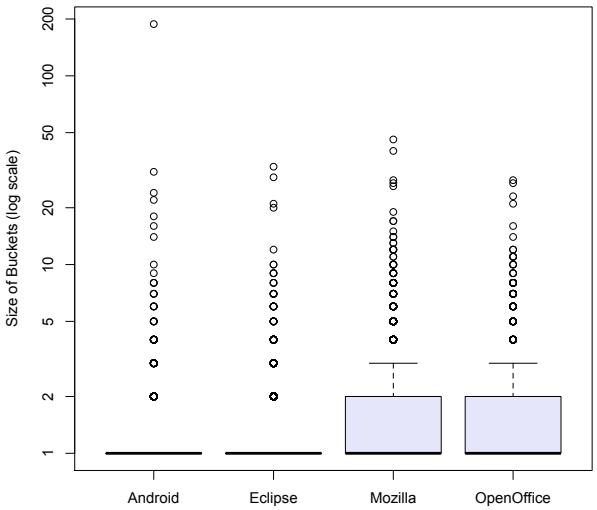


**Fig. 1** Distribution of Android, Eclipse, Mozilla, and OpenOffice duplicate bug reports into buckets.

**Table 2** Fields of Interest in Our Research

| Feature | Feature Type | Description |
|---|---|---|
| Summary | Text | A brief description of the problem. |
| Description | Text | A detailed declaration of the problem which may include reproduction steps and stack traces. |
| Product | Category | The product the report is about. |
| Component | Category | The component the report is about. |
| Version | Category | The version of the product the bug report is about. |
| Priority | Category | The priority of the report to be fixed. |
| Type | Category | The type of the report: defect, enhancement, task, feature. |
| Status | Category | The current status of the bug report: Fixed, Closed, Resolved, Duplicate, etc. |
| Merge ID | Identifier | If the report is a duplicate report, this field shows the ID of the report which the bug report is duplicating. |

recognized as a duplicate of bug report B by a triager, the Merge_ID of bug report A will be set to the ID of bug report B, and B will be the *immediate master* of A. Table 3 describes some examples of duplicate bug reports with their immediate master reports in Android bug-tracking system.

Table 3 shows examples of pairs of duplicate bug reports from Android and their categorical features. The *Product* field does not have any values in this table since Android bug reports do not have the *Product* field. Inconsistent use of fields, or differing fields across products are common challenges when addressing multiple bug repositories. The *Summary* and *Description* fields are not shown in this table.

**Table 3** Examples of duplicate bug reports from Android bug-tracking system.

| Pair | ID | Component | Product | 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 |

### 3.1 Software-Engineering Context in Bug Descriptions

To study the effect of leveraging software-engineering contexts on detecting duplicate bug reports, we represent context as textual similarity to a word list labelled with a context, such as maintainability or an architectural component. We have taken advantage of different existing software-related contextual data-sets represented as lists of contextual words. These contextual word lists are compared with the textual features of the bug reports to indicate the contextual characteristics of the bug reports, relating the bug-report words to existing contexts and thus annotate bug reports with context. Next, we describe the contextual word lists that we used.

- **Architecture words:** We manually created a set of architecture words for each bug repository. Each set is organized into a few word lists representing an architectural layer. All of our projects had some kind of architectural layering. Architectural words are often extracted from project documentation.
  For the Android bug repository, we utilized the word lists provided by Guana *et al.* [12]. They produced a set of Android architecture words to categorize Android bug reports based on the Android layered architecture. These words were extracted from Android architecture documents and are organized in five word lists (one word list per Android architectural layer [9]): Applications, Framework, Libraries, Runtime, and Kernel. Guana *et al.* spent less than 1 hour extracting these word lists.
  For the Eclipse bug repository, we manually created a set of architecture words [7], organized in three word lists (one word list per Eclipse architectural layer) with the following labels: IDE, Plugins, and Workbench. We spent less than 2 hours extracting these word lists.
  For the OpenOffice bug repository, the architectural words [21] were manually extracted and organized into four word lists: Abstract layer, Applications layer, Framework layer, and Infrastructure layer. We spent less than 2 hours extracting these word lists.
  The architectural words related to Mozilla Firefox software system were also manually extracted [11] and organized into five lists that describe architectural components and layers: Extensions, UI, Script, XPCOM, and Gecko. We spent less than 2 hours extracting these word lists.
- **Non-Functional Requirement (NFR) words:** Hindle *et al.* [16] proposed a method to automate labeled topic extraction, built upon LDA,

from commit-log comments in source-control systems. They labeled the topics from a generalizable cross-project taxonomy, consisting of non-functional requirements such as portability, maintainability, efficiency, etc. Their dataset of software NFR words is organized in six word lists with the following labels: *Efficiency, Functionality, Maintainability, Portability, Reliability*, and *Usability*. Some of the wordlists were automatically extracted from Wordnet, one set came from another study [10]. These word lists were used as the NFR context words in this work.

- **LDA topic words:** LDA represents the topic structure and topic relations among the bug reports. Two duplicate bug reports must address the same technical topics. The topic selection of a bug report is affected by the buggy topics for which the report is intended.
  Han *et al.* [13] applied both LDA and Labeled-LDA [26] topic analysis models to Android bug reports. We are using their Android HTC LDA topics, organized in 35 word-lists, $\text{Topic}_i$ where $i$ ranges from 0 to 34. These labelled topics took Han *et al.* 60 person-hours to extract. 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*.
  For Mozilla, Eclipse, and OpenOffice, we utilized the Vowpal Wabbit online learning tool [20] to extract the topics using LDA. For each of these repositories 20 topics were extracted (with parameters $\alpha = 0.1$, $\beta = 0.1$ and $N = 20$) then each topic's top 25 words were used as the word list for representing the corresponding topic. These word lists are labeled as Topic$i$ where $i \in 0...19$. LDA extraction took the authors 0.5 person-hours per product. Nguyen *et al.* [25] recommend choosing more than 100 topics and to explore different various topic settings.
- **Random English words:** To investigate the influence of contextual word lists on the accuracy of detecting duplicate bug reports, we created a collection of randomly selected English dictionary words. We created this "artificial context" to study if adding noise data to the features of the bug reports can improve or hamper 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 that start with the same English letter as the label of the word list. Random word lists took the authors 0.5 person-hours to automate.

Software engineering contexts can be built automatically or manually. In this work we present using LDA to generate contexts automatically, but many of our datasets are manually created and come from other studies. Effectively a reuse of existing contexts. Prior work [10, 19] describes in detail how to build contexts effectively. Nonetheless certain contexts, such as architectural contexts, can be constructed manually quite easily. If a product is split into packages, file names can and the package name itself can be put together into package contexts. Modules listed by within project documentation can be extracted as well. Currently the methodology for building contexts ranges from ad-hoc to more process oriented as suggested by Kayed *et al.* [19].

## 4 Methodology

In this Section, we describe our approach of duplicate bug-report identification. First, we explain our bug-report preprocessing approach. Next, we describe our similarity measurement method to compare the bug reports in terms of their textual, categorical and contextual characteristics. Then, we describe our duplicate bug report retrieval method based on our bug report similarity measurements. Finally, we present our evaluation approach to assess our duplicate bug report retrieval method.

### 4.1 Preprocessing

After extracting the bug reports, we apply a preprocessing method consisting of the following two steps.

1. The first step involves tokenizing the textual fields (*description* and *title*) of the bug reports and removing the stop words.
2. The second step involves the organization of the bug reports into a list of buckets. All the bug reports are inserted in the same bucket with their master bug report (specified by their Merge_ID). The bug report with the earliest open time becomes the master report of the bucket.

At the end of this process, the bug reports are converted into a collection of bug-report objects with the following properties: *Bug ID, description, title, status, component, priority, type, product, version, open date, close date,* and optional *master_id*, the ID of the bug report which is the master report of the bucket including the current bug report. Table 4 illustrates some examples of titles of Android bug reports before and after preprocessing.

### 4.2 Textual and Categorical Similarity Measurement

To compare the textual and categorical features of two bug reports, we measure their similarity based on their basic fields (*component, type, priority,*

**Table 4** Examples of Android bug reports before and after preprocessing

| Bug ID | Primitive Title | Processed Title |
|---|---|---|
| 3063 | Bluetooth does not work with Voice Dialer | bluetooth work voice dialer |
| 8152 | Need the ability to use voice dial over bluetooth | ability voice dial bluetooth |
| 3029 | support for Indian Regional Languages | support indian regional languages |
| 31989 | [ICS] Question of Google Maps' location pointer | ics question google maps location pointer |

*product* and *version*) shown in Table 2. Table 3 shows that duplicate bug reports frequently have similar categorical features, which motivates the use of categorical features in bug-deduplication. Figure 2 indicates the textual and categorical-similarity measurement formulas that we apply, adapted from Sun *et al.* [30].

Sun *et al.* [30] employed BM25F [37], a derivative of BM25 [27]. BM25 is used in search engines as a ranking function to rank query results. BM25 is much like TF-IDF except that it tunable and normalizable against average document length. It is loosely calculated by multiplying the inverse document frequency of a query term appearing in the underlying corpus by a document frequency normalized by 2 tunable constants $k_1$ and $b$, and document size. BM25F extends BM25 and makes it field or feature aware. BM25F extends BM25 with awareness of multiple textual fields of different sizes. Instead of normalizing by average document length, BM25F will normalize term frequencies per field by average field length. BM25F expands the BM25 $b$ parameter by the number of fields used (e.g. $b_f$ given field $f$), allowing one to optimize against different fields. In Sun *et al.* [30] BM25F is only used against 1 field at a time. Where as in Nguyen *et al.* [25] BM25F is used with multiple fields.

$$comparison_1(d_1, d_2) = BM25F(d_1, d_2) \quad \text{The comparison unit is unigram.}$$

$$comparison_2(d_1, d_2) = BM25F(d_1, d_2) \quad \text{The comparison unit is bigram.}$$

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

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

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

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

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

**Fig. 2** Categorical and textual measurements to compare a pair of bug reports [30].

The first comparison defined in Figure 2 is the textual-similarity measurement between two bug reports over the fields *title* and *description*, computed by $BM25F$. The second comparison is similar to the first one, except that the fields *title* and *description* are represented in bi-grams (a bi-gram consists of two consecutive words). The remaining five comparisons are categorical comparisons between reports.

$comparison_3$ compares the *product* of bug reports and does not apply to the Android bug repository, since Android bug reports do not specify a *product* feature. Therefore, we set the value of this feature to 0 for all Android bug reports. Also, regarding Sun *et al.*'s [30] method, we are not considering the version comparison for the bug reports of Eclipse, Mozilla, and OpenOffice bug repositories.

$Comparison_4$ 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, for example in Android bug-tracking system it shows whether they are both "defects" or "enhancements". This comparison has the value of 1 if the two bug reports are of 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). Priority is represented as the property $d.prio$ of document $d$. This is the priority set in the bug-tracking system. Thus $|d_1.prio - d_2.prio|$ describes the absolute difference in rank of priority values between 2 documents. And $d.vers$ describes the version number or rank of version of document $d$ in the bug-tracking system. $|d_1.vers - d_2.vers|$ describes the absolute difference in version number or rank [30]. If the two bug reports have similar priority or version this value will be 1. The greater the difference between priority and version, the higher the denominator and thus the closer to 0 the value for $Comparison_6$ and $comparison_7$.

The result of these comparisons establish a data-set including all the pairs of bug reports with the seven comparisons, shown in Figure 2, and a classification column, which reports whether the compared bug reports are duplicates of each other. Table 5 shows a snapshot of this data-set with some examples of pairs of Android bug reports. The value of class column is "dup" if the bug reports are in the same bucket and "non" otherwise.

**Table 5** Some examples of pairs of the bug reports from Android bug repository with categorical and textual similarity measurements ("textual_categorical" table).

| ID1 | ID2 | $BM25F_{un}$ | $BM25F_{bi}$ | Prod cmp | Comp cmp | Type cmp | Prio cmp | Vers cmp | Class |
|---|---|---|---|---|---|---|---|---|---|
| 14518 | 14516 | 1.4841 | 0.0000 | 0 | 1 | 1 | 1.0000 | 1.0000 | dup |
| 29374 | 3462 | 0.6282 | 0.1203 | 0 | 0 | 1 | 1.0000 | 1.0000 | non |
| 27904 | 14518 | 0.1190 | 0.0000 | 0 | 0 | 1 | 0.3333 | 0.1667 | non |

A huge number of pairs of bug reports is generated in this step. Thus, we conducted the evaluation of our method on the decision problem on a sample of the "textual_categorical" tables. There are very few pairs of bug reports marked as "dup" relative to the number of all the pairs ($\binom{size}{2}$, $size$ = total number of reports in the repository). Because we want to create a set of bug report pairs including 20% "dup"s and 80% "non"s, we randomly selected 4000 "dup" and 16000 "non" pairs of reports. Thus, for each bug repository, we produced 20000 sampled pairs of bug reports.

#### *4.2.1 Undersampling*

A short comment is worthwhile here to motivate our choice for testing with our method of sampling. The large sizes of the original data-sets implies substantial performance challenges in the training and evaluation of models. Furthermore the original class imbalance between unique and duplicate bugs is amplified by the pairing process. A class imbalance of 5% where 5% of bugs are duplicates and 95% are unique will cause a 99.75% class imbalance in terms of pairs. This means that any classifier who always returns the majority class will be 99.75% accurate (but with 0.0 Kappa). In this work we engage in undersampling.

Undersampling, *i.e.,* sampling the majority class at a lesser rate than the minority class, for logistic regression has been shown to be effective, as Wallace *et al.* [34] argue that, "For classification, this simple approach works well [33] and is theoretically motivated [35]." Thus, for some classifiers, undersampling during training results in a better classifier on the imbalanced dataset. Yet other classifiers examined in this work, such as both Naive Bayes and C4.5, are known to be insensitive to class imbalance [22].

80/20 undersampling enables us to train logistic-regression learners, and similar learners realistically with available memory and time and apply those trained models to larger datasets. Undersampling is the realistic training case for many of these machine learners. Undersampling also lets us better evaluate classification effectiveness, as the information gain from the learners becomes more apparent with less skewness.

80/20 was chosen to address that class imbalance does exist and we wanted to measure the effect of finding true-duplicates, thus we need lots of true-duplicates to test against. If we chose 90/10 or 99/1 we would need to use much larger samples to appropriately perceive changes in performance. In later work by Karan *et al.* [1], different proportions are tested achieving a similar effect.

In summary, our reasons for undersampling are for better logistic-regression training, better training time performance, ease of interpreting gains (or lack thereof) of learners under different treatments, and that the number of duplicate pairs are quickly washed out by $O(N^2)$ comparisons.

### 4.3 Contextual Similarity Measurement

As mentioned in Section 2, most of the previous research on detecting duplicate bug reports has focused on textual similarity measurements and IR techniques. Relatively few methods have also considered the categorical features of bug reports, and code segments or stack traces included with the description. In this section, we describe our approach for measuring the contextual similarity between bug reports. Considering the context of a bug report as a feature in the similarity-assessment process improves the accuracy of duplicate-bug detection.

In our method, we take advantage of the software contextual word lists described in Section 3. We explain the contribution of context in detail, using the NFR context as an example. As pointed out earlier, 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 query, and calculate the similarity between each query 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 6 shows the contextual features resulting from the comparison of the NFR context against some 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 are five different contextual word collections for the Android bug repository (Labeled-LDA, LDA, NFR, Android architecture, and English random words). And, there are four contextual word collection for each of the other bug repositories since they lack the Labeled-LDA contextual words.

**Table 6** Examples of the NFR contextual features for some of Android bug reports ("table of contextual measures")

| Bug ID | Efficiency | Functionality | Maintainability | Portability | Reliability | Usability |
|---|---|---|---|---|---|---|
| 3462 | 3.45 | 4.57 | 1.35 | 0.57 | 1.53 | 1.41 |
| 2282 | 2.88 | 2.51 | 1.07 | 3.37 | 4.53 | 4.91 |
| 29374 | 3.89 | 2.52 | 0.13 | 0.99 | 3.20 | 5.07 |
| 27904 | 2.93 | 1.03 | 0.50 | 0.00 | 3.36 | 4.55 |

### 4.4 Combining the Measurements

In this phase of the process, we have the "textual_categorical" table for pairs of bug reports (as shown in Table 5) and a number of tables reporting contextual-similarity measurements, each one according to a different context for individ-

ual bug reports, as described in Section 4.3. Here, we describe the combination of the "textual_categorical" table and the "tables of contextual measures".

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. An example of features and comparison for the "NFR" context is shown in Table 7 for the Android bug repository. In this table, the first seven columns are the same as the ones in Table 5; they report the similarity measurements between the two bug reports according to the 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 Table 7 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 vectors using the cosine similarity measurement, according to the formula shown below.

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

In this formula, $n$ is the number of word lists, *i.e.,* contextual features, for the particular context 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 the first and second bug reports respectively. The cosine similarity feature is demonstrated in Table 7. This table reports the comparison of bug reports with IDs 3462 and 2282, and bug reports with IDs 29374 and 3462, in terms of their NFR context. The first pair belongs to the same bucket (with class value of "dup"). The second pair of bug reports belong in different buckets (with the class value of "non"). Table 7 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.

**Table 7** Examples of the records in the data-set containing categorical, textual, and contextual measurements for the pairs of Android bug reports.

| ID1 | ID2 | $cmp_1$ | ... | $cmp_7$ | $Efficiency_1$ | ... | $Usability_1$ | $Efficiency_2$ | ... | $Usability_2$ | Cosine | Class |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3462 | 2282 | 1.52 | ... | 0.29 | 3.45 | ... | 1.41 | 2.88 | ... | 4.91 | 0.73 | dup |
| 29374 | 3462 | 0.63 | ... | 1.00 | 3.89 | ... | 5.07 | 3.45 | ... | 1.41 | 0.79 | non |

The class value (the classification) in the "all-features" tables should be predicted by the machine-learning classifiers in next phase. In other words, these classifiers decide whether the two bug reports are duplicates of each other.

## 4.5 Prediction

In this Section, we discuss two use cases related to the general task of duplicate-bug detection. The first use case refers to a basic decision problem, of whether two specific bug reports are duplicates of each other (given their similarity measurements). The second scenario reflects the general triaging scenario, where the incoming bug report is compared against all reports in the repository and a ranked list of candidate duplicates is presented to the triagers who can make the final decision about the real duplicates.

Clearly the two scenarios are closely related, since at their core they both make an assessment of how similar two bug reports are. Classification methods, deciding whether a bug is a duplicate of another bug or not, can be combined with ranked prediction, potentially to trim the ranked list. Classification is a much simpler process and we use it in this work to investigate differences between the different feature-sets and techniques. Ranked lists are not usually evaluated on bug reports without duplicates. One problem with both of scenarios is that fundamentally there are $O(N^2)$ comparisons and with larger values of $N$ like say $N > 80000$ there are over 3 billion comparisons to be made. As a result neither scenario scales well. Smart indexing can be used to improve the ranked list problem but it might miss duplicates.

More generally, classification is appropriate:

- when the question asked is "are these two bug reports duplicates?";
- when the bug repository is small;
- when one is interested in duplicates of a single bug only and can spend $O(n)$ time searching for them;
- when one is exploring the effect of new information when searching for duplicate reports; and
- when one wants to explore false negatives, true negatives and false positives.

On the other hand, a ranked list is preferable:

- when the question is "give me a list of possible duplicate bug reports ranked by potential duplicate status?";
- when a human triager is manually searching for candidate duplicate bugs; and
- when order or efficiency of evaluation (reading bugs) matters.

### *4.5.1 Classification*

In this Section, we discuss the application of classifiers on different sets of our comparison metrics for deciding whether a pair of bug reports are duplicates or not. The idea is to use machine learning to amplify the impact of the work of the triager; as the triager identifies duplicate bugs, the classifier learns how to better recognize duplicates and may suggest candidates to the triager and thus simplify his/her task.

To retrieve the duplicate bug reports we take advantage of several 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 classifiers. Each "all-features" table (described earlier) includes all the inputs necessary for the classifiers to classify each pair of bug reports as duplicate or not. To avoid over fitting during training and evaluation, we use the 10-fold cross validation technique.

The classifiers we use are implemented by Weka (Weikato Environment for Knowledge Analysis) [17]. Weka is a well-known machine learning tool implemented in Java. This tool supports numerous data mining tasks such as preprocessing, clustering, classification, regression, visualization, and feature selection. Weka accepts data in a flat file in which each instance has a fixed number of attributes. More specifically, we use the 0-R algorithm to establish the baseline, C4.5, K-NN (K Nearest Neighbours), Logistic Regression, and Naive Bayes. K-NN tends to perform well with many features, and when it works we can infer that the input data has a fundamentally simple structure that is exploitable by distance measures.

*Evaluation Metrics*

To evaluate the performance of the classifiers to classify bug-report pairs as duplicates or not, we use 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 below.

$$acc = \frac{|true\ dup| + |true\ non|}{|true\ dup| + |false\ dup| + |true\ non| + |false\ non|} \tag{3}$$

True "dup" and false "dup" are the pairs of bug reports truly and wrongly recognized as "dup" respectively by classifiers. True "non" and false "non" have the same definition but for the "non" class value.

Kappa is a statistical measure for inter-rater agreement. For example, kappa demonstrates how much homogeneity there is in the rating given by raters. The equation for kappa is

$$kappa = \frac{Pr(a) - Pr(e)}{1 - Pr(e)} \tag{4}$$

$Pr(a)$ is the relative observed agreement among the raters. $Pr(e)$ is the hypothetical probability of chance agreement, using the observed data, to calculate the probabilities of each observer randomly saying each category. If the judges are in complete agreement, then $kappa = 1$. If there is no agreement among them other than what would be expected by chance, then $kappa = 0$. We use Kappa to assess the agreement between our gold standard and the machine learning classifier.

AUC is the area under the Receiver Operation Characteristic (ROC) curve. The ROC curve is created by plotting the fraction of truly recognized "dup"s out of all the recognized "dup"s (True Positive Rate) versus the fraction of wrongly recognized "dup"s out of all the recognized "non"s (False Positive Rate) by the classifiers. 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).

*4.5.2 Retrieving the List of the Most Similar Candidates*

In this Section we discuss the application of our method to the second duplicate-bug detection scenario, namely that of selecting a ranked list of candidate duplicates for the triager to inspect more closely. Figure 3 demonstrates the method of retrieving top-k similar bug reports to a specific incoming report. As indicated in this figure, there is a similarity criterion, which compares the incoming bug report against the existing reports in the repository and returns a sorted list of candidate duplicate reports. To compare the bug reports contextually, we have proposed three similarity criteria: (1) Cosine similarity based metric; (2) Euclidean distance based metric; and (3) Logistic regression based metric.

As discussed in Section 3, Sun *et al.* [30] applied a linear function called REP to sort the candidate masters, which indicated promising results in detecting correct masters for duplicate reports. Therefore, we decided to combine our contextual comparison metrics with REP so that we can take advantage of all the textual, categorical, and contextual features when comparing bug reports.

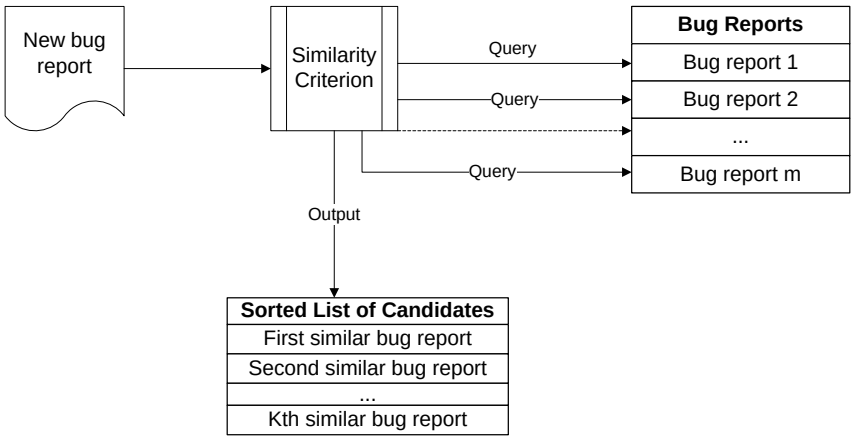


**Fig. 3** Overall workflow to retrieve duplicate bug reports

To implement this method, we have constructed a data-set including all the pairs of bug reports for each bug repository. This data-set includes all the

pairs with the similarity criteria we would like to study in the experiment at hand (the "similarity_criteria" data-set). For instance, if the only similarity criterion is the REP function (Sun *et al.*'s method [30]), the data-set includes the IDs of the bug reports, the label for each pair ("dup" or "non") , and the REP result for each pair. Table 8 indicates some sample records from the "similarity_criteria" data-set with the REP criterion for Mozilla bug reports.

**Table 8** Examples of pairs of bug reports from Mozilla bug repository with their REP comparisons result and their class (the "similarity_criteria" table)

| Bug ID 1 | Bug ID 2 | REP | Class |
|---|---|---|---|
| 563260 | 576854 | 2.617 | dup |
| 563250 | 596269 | 3.388 | non |
| 563325 | 612618 | 0.095 | non |
| 563308 | 582608 | 1.928 | non |
| 563276 | 602852 | 0.576 | non |

*Cosine Similarity* The first similarity metric is the cosine similarity for a specific context, shown in Equation (2). In this formula, the contextual weight vectors of the two bug reports are compared. If two bug reports were exactly similar in terms of a particular context, their contextual cosine similarity is equal to 1. And, if the two bug reports were completely different in terms of that context, this value is 0.

Since we expect this similarity criterion to assign a higher score to the more similar reports in comparison to the non-similar ones, we utilize it as a duplicate report retrieval criterion. This metric returns higher values when it compares more similar vectors. To combine the cosine similarity and REP we normalize REP to scale it in the range of 0 to 1 and we then calculate their average as indicated below:

$$combined\_cosine\_metric(B_1, B_2) = \frac{norm(REP(B_1, B_2)) + cosine\_sim(C_1, C_2)}{2} \quad (5)$$

In this function $C_1$ and $C_2$ represent the contextual features of the bug reports $B_1$ and $B_2$ respectively. As an example, for the NFR context, $C_1$ and $C_2$ contain six dimensions each. Equation (5) combines all the textual, categorical, and contextual similarity metrics to compare a pair of bug reports. In some experiments only the $cosine\_sim(C_1, C_2)$ formula is applied to compare the bug reports only contextually.

To apply this criterion, and its combination with the REP function, we have added some new comparison metrics to the "similarity_criteria" data-set. Table 9 shows some sample records from the "similarity_criteria" data-set with the $REP$, and $cosine\_sim$ criteria (for different contexts) from Mozilla bug reports. So, if we want to combine the REP and $cosine\_sim$ functions, we exploit two columns from this table, otherwise we only utilize one of them to compare the bug reports.

**Table 9** Examples of pairs of bug report from Mozilla repository with their REP and cosine_sim comparisons for different contexts and their class

| Bug ID1 | Bug ID2 | REP | Architecture cosine | NFR cosine | Random cosine | LDA cosine | Class |
|---|---|---|---|---|---|---|---|
| 563260 | 576854 | 2.617 | 0.622 | 0.625 | 0.000 | 0.807 | dup |
| 563250 | 596269 | 3.388 | 0.955 | 0.877 | 0.000 | 0.392 | non |
| 563325 | 612618 | 0.095 | 0.000 | 0.854 | 0.474 | 0.076 | non |
| 563308 | 582608 | 1.928 | 0.256 | 0.916 | 0.000 | 0.077 | non |
| 563276 | 602852 | 0.576 | 0.000 | 0.802 | 0.000 | 0.000 | non |

*Euclidean Distance*

The second similarity metric is established based on the Euclidean distance between two context vectors corresponding to the two bug reports being compared, as shown in the following formula.

$$contextual\_distance(B_1, B_2) = \sum_{i=1}^{n} \frac{1}{1 + |C_{1i} - C_{2i}|} \quad (6)$$

In this function, $n$ is the number of the word lists of the context at hand (such as 6 for NFR context). $C_1$ and $C_2$ are the contextual features for the bug reports $B_1$ and $B_2$ respectively. This similarity metric is analogous to the priority and version comparison metrics illustrated in Figure 2. In this function, as the distance between the two context vectors increases, the resulting value approaches to 0. And, when this distance decreases, the resulting value approaches to 1. Therefore, this function results in higher scores for the bug reports contextually close to each other.

To compare the bug reports textually, categorically, and contextually (utilizing the above contextual similarity metric) we have combined Equation (6) and the REP linear function. To that end, we simply added the contextual-distance function to REP, which could be considered as adding a few more features like priority and version to the REP function. The formula is shown below:

$$combined\_euclidean\_metric(B_1, B_2) = REP(B_1, B_2) + contextual\_distance(B_1, B_2) \quad (7)$$

In the above formula, $B_1$ and $B_2$ are the two bug reports being compared. To apply this criterion, we have added some new comparison metrics to the "similarity_criteria" data-set (demonstrated in Table 8). Table 10 indicates some sample records from this data-set with the REP, and *contextual_distance* criteria (for different contexts) from Mozilla bug reports.

*Logistic Regression*

We chose logistic regression to combine REP and contextual comparison metrics, because of its reasonable performance and because it is straightforward to configure with appropriate coefficients using a probabilistic model. Logistic regression effectively provides us with a ranking function for pairs.

**Table 10** Examples of pairs of bug reports from Mozilla repository with their REP and contextual-distance comparisons for different contexts and their class

| Bug ID1 | Bug ID2 | REP | Arch dist. | NFR dist. | Random dist. | LDA dist. | Class |
|---|---|---|---|---|---|---|---|
| 563260 | 576854 | 2.6170 | 2.2987 | 3.3549 | 0.9741 | 10.8501 | dup |
| 563250 | 596269 | 3.3880 | 1.6458 | 3.1567 | 1.1871 | 09.6204 | non |
| 563325 | 612618 | 0.0950 | 0.5789 | 1.9480 | 1.4071 | 10.3120 | non |
| 563308 | 582608 | 1.9280 | 1.3722 | 3.4686 | 1.1337 | 05.3772 | non |
| 563276 | 602852 | 0.5760 | 0.8739 | 2.5228 | 2.5519 | 02.5027 | non |

This similarity criterion works based on the logistic-regression classifier. This classifier is applied in cases where the observed outcome (dependent variable) can accept only two possible values (in our case "dup" or "non"). In this classifier, the probability function (of the bug being a duplicate) is modeled as a linear function which includes a linear combination of the independent variables and a set of estimated coefficients. The predictor function for a particular data point i is written as follows:

$$f(i) = \beta_0 + \beta_1 xi1 + \beta_2 x_{2i} + ... + \beta_m x_{mi} \tag{8}$$

in which $\beta_0 - \beta_m$ are regression coefficients indicating the impact of each particular descriptive variable on the outcome.

In our approach, we have taken advantage of the prediction function (8) to establish a similarity criterion based to sort the candidate duplicates with. Considering the REP and the *cosine_sim* (Equation (2)) functions as the descriptive variables, we are interested in distinguishing the appropriate coefficients for them in the similarity criterion. In other words, we would like to discover improved coefficients for the *cosine_sim* and REP measures to combine them, rather than simply calculating their average (such as what is done in Equation (5)). To that end, we have sampled the "similarity_criteria" data-set including the REP and *cosine_sim* similarity functions (illustrated in Table 9). The sampled data-set includes 10000 records with 20% of records labeled as "dup" and 80% of records labeled as "non". Then, the logistic regression classifier is applied on this sampled data-set. If we intend to exploit the REP function exclusively in our similarity criterion, we apply the logistic regression on the sampled data-set including only the REP similarity metric; the resulting similarity criterion, with coefficients are estimated by the logistic regression classifier, is as follows:

$$criterion = \beta_0 + \beta_1(REP) \tag{9}$$

Furthermore, when we aim to combine two different metrics such as REP and cosine_sim for the NFR context to sort the candidate duplicates, we apply the logistic regression on the sampled data-set including the REP and the cosine_sim for NFR to predict the coefficients. The similarity criterion is defined as follows:

$$criterion = \beta_0 + \beta_1(REP) + \beta_2(cosine\_sim_{NFR}) \tag{10}$$

Also, if we aim to investigate the effect of contextual comparison (for a specific context such as NFR) on duplicate bug report retrieval, the similarity criterion changes as follows:

$$criterion = \beta_0 + \beta_1(cosine\_sim_{NFR}) \tag{11}$$

These criteria are in fact the prediction function described in Equation (8) that are exploited as the bug report comparison function.

*Evaluating the List of Candidates*

So far, we have presented our contextual methods for duplicate-bug detection. In this Section, we focus on the technique we exploit to assess the retrieval approach, which compares every incoming bug report against all the existing reports in the repository and sorts the existing bug reports based on their similarity to the incoming report. The list of retrieved candidates is sorted in descending order of similarity, with the more similar reports ranked higher.

The quality of the retrieval process is measured by studying the indices of the true duplicates (of the incoming report) in the list. The higher in the list these true duplicates are, the better we conclude the similarity criterion of the retrieval process to be. In our study, we evaluate the sorted list of similar reports using the *Mean Average Precision* (MAP) measure. MAP is a scalar-valued measure of ranked retrieval results, designed to evaluate the performance of queries that can have multiple relevant answers. In our duplicate bug-report retrieval method, each bug report can have several duplicates since there may be several bug reports in a bucket. Given $Q$ duplicate bug reports, for each of them, the system retrieves potential duplicates in descending order of similarity (until all the duplicates of the bug report are retrieved) and records their indexes in the sorted list. The MAP measure is calculated as follows:

$$MAP = \frac{\sum_{q=1}^{Q} AvgP(q)}{Q} \tag{12}$$

Buckley *et al.* [8] define average precision as "The mean of the precision scores obtained after each relevant document is retrieved, using zero as the precision for relevant documents that are not retrieved." $AvgP(q)$ where the number of relevant documents is 0 is equal to 0, otherwise $AvgP(q)$ is defined as:

$$AvgP(q) = \frac{\sum_{k=1}^{n} P(k) \times (Rel(K))}{number\_of\_relevant\_documents} \tag{13}$$

In the above functions, relevant_documents are the actual duplicates; $Rel(K) = 1$ when the documents are duplicates of each other and $Rel(K) = 0$ otherwise; $n$ is the number of bug reports in the repository; and $p(k)$ is the precision at the cut-off $k$. The precision function is provided below:

$$precision = \frac{|\{relevant \quad docs\} \bigcap \{retrieved \quad docs\}|}{|retrieved \quad docs|} \tag{14}$$

Since in our experiments we want to evaluate the retrieved list of candidates, we should know the actual duplicates of an incoming report. Therefore, in our experiments, the incoming reports consist of all the reports marked as duplicate in the repository (that have specific masters). This approach is similar to the work of Sun *et al.* [30], who, however, have simplified MAP to an MRR-like measure discussed in Section 2. We believe that MAP is more appropriate than MRR for the candidate-duplicates retrieval task, since it is a proxy for the positions of all the true duplicates retrieved in the list and not only the position of the first true duplicate.

Sun *et al.* suggest that the first case that is in the same duplicate bucket is the right answer. But this assumes that the answers are all correct and that duplicate buckets have already been established and observed. A more realistic use of ranked lists is to read down the list and look for duplicate candidates and evaluate them. We should try to return a coherent list of bugs with duplicates in that list. A list with the correct duplicate deep down in the list is not as helpful. Thus MAP is an appropriate measure because when querying for a duplicate bug there is often more than 1 right answer, and furthermore the right answers to duplicate bugs might not have been marked or observed yet. MRR assumes we were capable of marking all of such bugs which is not realistic. Furthermore MRR has a positive bias where poorly ranked dupes are ignored at the expense of the top ranked duplicate. MAP does not have this bias and punishes search results that do not discover the majority of duplicate issues earlier. Both MRR and MAP suffer as performance measure when a user does not evaluate all returned candidates. In both cases MAP and MRR are still positive because they are not evaluated on negative queries: non-duplicates.

*The FastREP Algorithm*

In order to implement the logistic-regression based retrieval function and evaluate it with the MAP measure, we propose the "FastREP" algorithm. "FastREP" is designed to enable the calculation of one or many queries relatively quickly using a single linear scan of a table of comparisons. It involves the following steps.

1. As we discussed in Section 4.5.2, we sample the "similarity_criteria" data-set including the REP and *cosine_sim* similarity criteria (illustrated in Table 9). The sampled data-set includes 10000 records with 20% of records labeled as "dup" and 80% of records labeled as "non". Then, we apply the logistic-regression classifier on this sampled data-set. Depending on the experiment, the features involved in this classification may vary (the features may be only REP, only the contextual feature(s), or a combination of both).
2. Based on the coefficients returned by the logistic-regression classifier, the criterion function is built and the value of this function for each record in the entire "similarity_criteria" data-set (not the sampled data-set) is cal-

culated (applying a criterion function similar to either one of Equation (9), (10). This value is added as a new column in the table.
3. The resulting table that includes the criterion column is then sorted based on the values of the criterion column in descending order.
4. Next, a map data-structure is constructed that maps a bug report to a tuple of 3 numbers: $(seen, hits, sum)$. $seen$ is the number of pairs of bug report that have been observed. $hits$ is the number of duplicates of this specific bug report that have been observed. $sum$ is the sum of precision at $k$ calculations (meant to calculate average precision later) made each time a duplicate bug report is seen where $k$ is $seen$.
5. The sorted table of pairs is scanned. Each row is used to populate the map data-structure. For each row, if the pair references a known "dup"/duplicate bug report then $seen$ is incremented in the map for each of the bug reports in the pair. If the row represents a duplicate bug report pair (marked as "dup") then $hits$ is incremented and precision at k, $hits/seen$, is added to $sum$.
6. Next, the average precision (indicated in Equation (13)), $AvgP$, can be calculated for each "dup" bug report by dividing the $sum$ of each bug report by their $hits$.
7. Finally, the mean of all the $AvgP$s can be calculated to report the MAP result.

This method is fast because we can answer many queries in parallel and only a single sort step is required. The run-time is $O(NlogN)$ as we use a merge sort (GNU sort) to sort the large tables where $N$ is the number of comparisons.

## 5 Case Studies

We applied our method on bug reports from the Android, Eclipse, Mozilla, and OpenOffice bug-tracking systems. To study the effect of contextual data on the accuracy of duplicate bug-report detection, we applied the classification algorithms (mentioned in Section 4) on the Android bug repository in our recent work [2]. In this paper, we applied the same approach on Eclipse, Mozilla, and OpenOffice bug repositories as well. We applied classification algorithms on three different data-sets extracted from each bug-tracker:

1. the data-set including all of the similarity measurements illustrated in the "all-features" tables (such as Table 7, the NFR "all-features" table);
2. the data-set including only the textual and categorical similarity measurements of the bug reports; and
3. the data-set including only the contextual similarity measurement features.

As mentioned before, these data-sets include 20000 pairs of randomly selected bug reports with 20% "dup"s and 80% "non"s.

In addition to the experiments in our recent work [2], we have also conducted some experiments to provide the list of candidate duplicates benefiting

from the REP function presented by Sun *et al.* [30]. We have extended the REP function (in section 4.5.2) to apply our contextual approach when providing the list of candidates by three different methods: (1) cosine similarity based metric, (2) euclidean distance based metric, (3) logistic regression based metric. We discuss the experiments applying these retrieval methods in Section 5.4.

## 5.1 Evaluating the Classification-based Retrieval Method

In this Section we analyze the effect of context on detecting duplicate bug reports based on the results reported by the machine-learning classifiers while applying them on the "all_features" data-sets (described in Section 4) with and without contextual features. Here, we are eager to know the answer to the following question: *Does the software context improve the duplicate bug reports retrieval using the machine learning classifiers?*

Tables 11, 12, 13, and 14 show the statistical evaluation measurement values without considering the context of bug reports at the top. This part of the tables demonstrates the resulting evaluation measures of the machine learners with the input data created by Sun *et al.*'s method [30]. The maximum values are shown in bold. These tables demonstrate that Sun *et al.*'s method definitely finds duplicates.

Tables 11, 12, 13, and 14 also report the statistical measurement results, using the contextual data-sets in bug-report similarity measurements. The highest value in each column is again shown in bold.

As demonstrated in our previous work [2], classification algorithms can effectively identify the duplicate bug reports of the Android bug repository. Table 11 reports the results of applying these algorithms on this repository. As shown in this table, the highest improvements are achieved by utilizing the LDA and Labeled-LDA contextual data.

Tables 12 and 13 report the results for the Eclipse and Mozilla bug repositories respectively. As demonstrated in these tables, the LDA context result again in the highest improvement, however this improvement is trivial. According to tables 12 and 13, the contextual features exclusively improve the accuracy of detecting duplicate bug reports by around 6% over the baseline. 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. For the OpenOffice bug repository, as indicated in Table 14, the highest improvement (which is still trivial) is achieved by using the NFR context. The NFR contextual word lists are project independent so it can be considered as an automatic method of bug-report deduplication as well.

Table 15 shows some examples of predictions made by the K-NN algorithm for the Android bug repository, including textual, categorical, and Labeled-LDA context data. The first pair of bug reports is correctly recognized as duplicates, given that both of the reports are about "Bluetooth" (which is an

**Table 11** Statistical measures resulted by the experiments on Android bug repository including textual, categorical, and contextual data

| Context | Algorithm | Textual, Categorical, & Contextual | | | Contextual only | | |
|---|---|---|---|---|---|---|---|
| | | Accuracy | Kappa | AUC | Accuracy | Kappa | AUC |
| No Context | 0-R | 80.000% | 0.0000 | 0.500 | | | |
| | Logistic Regression | 82.830% | 0.3216 | 0.814 | | | |
| | Naive Bayes | 78.625% | -0.0081 | **0.778** | | | |
| | C4.5 | **84.525%** | 0.4324 | 0.716 | | | |
| | K-NN | 82.380% | **0.4616** | 0.737 | | | |
| Architecture | 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 |
| NFR | 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 |
| Random English Words | 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 |
| LDA | 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 |
| Labeled LDA | 0-R | 80.000% | 0.0000 | 0.500 | 80.000% | 0.0000 | 0.500 |
| | Logistic Regression | 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 |

Android Labeled-LDA topic). For the same reason the fourth pair is recognized as a duplicate, while the reports in this pair are not duplicates of each other. In the second pair, the bug reports are categorically different and also textually not similar in terms of the Android Labeled-LDA topics, but they are wrongly classified as non-duplicates by the machine learner. In the third pair, the reports are categorically similar but they are correctly recognized as non-duplicates as they are about two different Android Labeled-LDA topics.

Figure 4 shows the ROC curves for results of applying K-NN algorithm on various "all-features" tables (such as Table 7) for the Android bug repository. It also displays the ROC curve for the "textual_categorical" table (such as Table 5). The figure shows that the Labeled-LDA context outweighs the other

**Table 12** Statistical measures resulted by the experiments on Eclipse bug repository including textual, categorical, and contextual

| Context | Algorithm | Textual, Categorical, & Contextual | | | Contextual only | | |
|---|---|---|---|---|---|---|---|
| | | Accuracy | Kappa | AUC | Accuracy | Kappa | AUC |
| No Context | 0-R | 80.000% | 0.0000 | 0.500 | | | |
| | Logistic Regression | 96.610% | 0.8922 | **0.989** | | | |
| | Naive Bayes | 96.500% | 0.8896 | 0.985 | | | |
| | C4.5 | **96.650%** | **0.8947** | 0.975 | | | |
| | K-NN | 95.270% | 0.8522 | 0.915 | | | |
| Architecture | 0-R | 80.000% | 0.0000 | 0.500 | 80.000% | 0.0000 | 0.500 |
| | Logistic Regression | 96.685% | 0.8947 | 0.989 | 82.100% | 0.2164 | 0.718 |
| | Naive Bayes | 96.125% | 0.8786 | 0.983 | 77.660% | 0.2157 | 0.648 |
| | C4.5 | 96.700% | 0.8961 | 0.966 | 83.720% | 0.3462 | 0.700 |
| | K-NN | 94.395% | 0.8240 | 0.917 | 80.910% | 0.3852 | 0.714 |
| NFR | 0-R | 80.000% | 0.0000 | 0.500 | 80.000% | 0.0000 | 0.500 |
| | Logistic Regression | 96.680% | 0.8943 | 0.989 | 79.960% | 0.0337 | 0.665 |
| | Naive Bayes | 96.350% | 0.8848 | 0.980 | 79.960% | 0.0269 | 0.643 |
| | C4.5 | 96.585% | 0.893 | 0.955 | 83.130% | 0.3495 | 0.705 |
| | K-NN | 93.725% | 0.8043 | 0.904 | 78.010% | 0.3619 | 0.699 |
| Random English Words | 0-R | 80.000% | 0.0000 | 0.500 | 80.000% | 0.0000 | 0.500 |
| | Logistic Regression | 96.605% | 0.8921 | 0.989 | 80.720% | 0.0983 | 0.661 |
| | Naive Bayes | 92.095% | 0.7714 | 0.949 | 41.870% | 0.0702 | 0.610 |
| | C4.5 | 96.660% | 0.8954 | 0.964 | 83.120% | 0.3132 | 0.681 |
| | K-NN | 94.920% | 0.8417 | 0.930 | 80.600% | 0.3459 | 0.710 |
| LDA | 0-R | 80.000% | 0.0000 | 0.500 | 80.000% | 0.0000 | 0.500 |
| | Logistic Regression | **96.750%** | **0.8968** | **0.990** | **86.215%** | 0.4716 | 0.854 |
| | Naive Bayes | 94.710% | 0.8382 | 0.972 | 78.765% | 0.3208 | 0.722 |
| | C4.5 | 96.640% | 0.8945 | 0.954 | 85.120% | 0.5174 | 0.747 |
| | K-NN | 94.225% | 0.8222 | 0.919 | 84.070% | **0.5376** | **0.792** |

ones. The "No context" curve shows the performance of the 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, adding extra features with or without Sun *et al.*'s features improves bug-deduplication performance.

Figure 5 demonstrates the ROC curves for results of applying the C4.5 algorithm on the "all-features" tables for the Android bug repository. 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.

Figure 6 displays the ROC curves extracted by applying K-NN on "all-features" and "textual_categorical" tables for the Eclipse bug repository. In this diagram, the LDA context shows the highest improvement in performance. Figure 7 depicts the same curves extracted by applying the Logistic Regression algorithm. This diagram indicates a very slight improvement when applying the LDA context.

**Table 13** Statistical measures resulted by the experiments on Mozilla bug repository including textual, categorical, and contextual data

| Context | Algorithm | Textual, Categorical, & Contextual | | | Contextual only | | |
|---|---|---|---|---|---|---|---|
| | | Accuracy | Kappa | AUC | Accuracy | Kappa | AUC |
| No Context | 0-R | 80.000% | 0.0000 | 0.500 | | | |
| | Logistic Regression | 94.065% | 0.8075 | **0.971** | | | |
| | Naive Bayes | 92.670% | 0.7679 | 0.961 | | | |
| | C4.5 | **94.085%** | **0.8114** | 0.943 | | | |
| | K-NN | 92.810% | 0.7432 | 0.857 | | | |
| Architecture | 0-R | 80.000% | 0.0000 | 0.500 | 80.000% | 0.0000 | 0.500 |
| | Logistic Regression | 94.340% | 0.8169 | 0.973 | 80.450% | 0.0980 | 0.719 |
| | Naive Bayes | 91.105% | 0.7326 | 0.950 | 74.775% | 0.1248 | 0.646 |
| | C4.5 | 94.470% | 0.8247 | 0.938 | 84.985% | 0.4656 | 0.750 |
| | K-NN | 91.895% | 0.7451 | 0.879 | 82.395% | 0.4290 | 0.728 |
| NFR | 0-R | 80.000% | 0.0000 | 0.500 | 80.000% | 0.0000 | 0.500 |
| | Logistic Regression | 94.210% | 0.8133 | 0.973 | 79.825% | 0.0136 | 0.637 |
| | Naive Bayes | 92.650% | 0.7690 | 0.956 | 80.145% | 0.0380 | 0.658 |
| | C4.5 | 93.630% | 0.7968 | 0.914 | 80.285% | 0.1740 | 0.650 |
| | K-NN | 88.260% | 0.6343 | 0.818 | 73.465% | 0.2263 | 0.621 |
| Random English Words | 0-R | 80.000% | 0.0000 | 0.500 | 80.000% | 0.0000 | 0.500 |
| | Logistic Regression | 94.390% | 0.8188 | 0.973 | 82.110% | 0.1994 | 0.684 |
| | Naive Bayes | 78.075% | 0.4870 | 0.893 | 35.880% | 0.0461 | 0.638 |
| | C4.5 | 94.170% | 0.8151 | 0.941 | 82.635% | 0.2829 | 0.640 |
| | K-NN | 90.440% | 0.7020 | 0.859 | 79.620% | 0.3473 | 0.694 |
| LDA | 0-R | 80.000% | 0.0000 | 0.500 | 80.000% | 0.0000 | 0.500 |
| | Logistic Regression | **94.775%** | **0.8318** | **0.975** | 84.280% | 0.4049 | **0.823** |
| | Naive Bayes | 91.265% | 0.7331 | 0.947 | 78.855% | 0.3333 | 0.753 |
| | C4.5 | 94.135% | 0.8138 | 0.900 | **85.980%** | **0.5257** | 0.747 |
| | K-NN | 89.505% | 0.6796 | 0.849 | 83.415% | 0.5133 | 0.775 |

Figure 8 displays the ROC curves extracted by applying C4.5 on "all-features" and "textual_categorical" tables for the Mozilla bug repository. As indicated in this diagram, the LDA context shows the highest improvement. Figure 9 reveals the same curves extracted by applying the K-NN algorithm. As illustrated in this diagram, the highest improvement is achieved by the architecture context.

Figure 10 displays the ROC curves extracted by by applying C4.5 on "all-features" and "textual_categorical" tables for the OpenOffice bug repository. As this diagram shows, the highest performance is achieved by applying the LDA context. Figure 11 reveals the same curves extracted by applying the Logistic Regression algorithm. This diagram indicates a slight improvement when applying the NFR context.

**Table 14** Statistical measures resulted by the experiments on OpenOffice bug repository including textual, categorical, and contextual data

| Context | Algorithm | Textual, Categorical, & Contextual | | | Contextual only | | |
|---|---|---|---|---|---|---|---|
| | | Accuracy | Kappa | AUC | Accuracy | Kappa | AUC |
| No Context | 0-R | 80.000% | 0.0000 | 0.500 | | | |
| | Logistic Regression | 93.125% | 0.7729 | **0.961** | | | |
| | Naive Bayes | 91.415% | 0.6960 | 0.951 | | | |
| | C4.5 | **93.210%** | **0.7789** | 0.932 | | | |
| | K-NN | 90.580% | 0.7042 | 0.812 | | | |
| Architecture | 0-R | 80.000% | 0.0000 | 0.500 | 80.000% | 0.0000 | 0.500 |
| | Logistic Regression | 93.210% | 0.7761 | 0.961 | 80.000% | 0.0000 | 0.600 |
| | Naive Bayes | 91.545% | 0.7039 | 0.938 | 80.000% | 0.0000 | 0.604 |
| | C4.5 | 92.975% | 0.7734 | 0.920 | 79.995% | 0.0565 | 0.573 |
| | K-NN | 88.400% | 0.6332 | 0.821 | 77.180% | 0.1926 | 0.647 |
| NFR | 0-R | 80.000% | 0.0000 | 0.500 | 80.000% | 0.0000 | 0.500 |
| | Logistic Regression | **93.670%** | **0.7925** | **0.966** | 80.010% | 0.0011 | 0.624 |
| | Naive Bayes | 91.470% | 0.6988 | 0.947 | 80.000% | 0.0000 | 0.604 |
| | C4.5 | 92.380% | 0.7562 | 0.893 | 80.325% | 0.1775 | 0.645 |
| | K-NN | 84.105% | 0.5130 | 0.762 | 73.605% | 0.2114 | 0.611 |
| Random English Words | 0-R | 80.000% | 0.0000 | 0.500 | 80.000% | 0.0000 | 0.500 |
| | Logistic Regression | 93.275% | 0.7803 | 0.961 | 80.550% | 0.1045 | 0.631 |
| | Naive Bayes | 83.460% | 0.5669 | 0.880 | 31.860% | 0.0296 | 0.591 |
| | C4.5 | 93.120% | 0.7762 | 0.935 | 81.215% | 0.1657 | 0.586 |
| | K-NN | 88.015% | 0.6266 | 0.809 | 78.605% | 0.2808 | 0.692 |
| LDA | 0-R | 80.000% | 0.0000 | 0.500 | 80.000% | 0.0000 | 0.500 |
| | Logistic Regression | 93.385% | 0.7825 | 0.964 | **81.030%** | 0.1435 | **0.699** |
| | Naive Bayes | 89.090% | 0.6402 | 0.911 | 74.870% | 0.1551 | 0.630 |
| | C4.5 | 92.505% | 0.7618 | 0.890 | 79.350% | 0.2502 | 0.654 |
| | K-NN | 84.130% | 0.5111 | 0.760 | 77.935% | **0.3507** | 0.688 |

**Table 15** Examples of predictions made by K-NN algorithm for Android bug repository including textual, categorical, and Labeled-LDA context's data

| Pair | Title | Comp. | Prio. | Type | Vers. | Act. | Pred. |
|---|---|---|---|---|---|---|---|
| 1 | Bluetooth does not work with Voice Dialer | Device | Med | Def. | | dup | dup |
| | Need the ability to use voice dial over bluetooth | | Med | Def. | | | |
| 2 | support for Indian Regional Languages... | Framework | Med | Enh. | | dup | non |
| | Indic fonts render without correctly reordering.. | GfxMedia | Med | Def. | | | |
| 3 | Bluetooth Phonebook Access Profile ... | | Med | Def. | 2.2 | non | non |
| | [ICS] Question of Google Maps' location pointer | | Med | Def. | | | |
| 4 | enhanced low-level Bluetooth support | Device | Med | Enh. | | non | dup |
| | Bluetooth DUN/PAN Tethering support | Device | Med | Enh. | | | |

## 5.2 Discussion of Findings

The results of our experimentation indicate that the contextual features improve the identification of duplicate bug reports, in some cases by a remarkable margin. Hence, we have established that the contextual information can

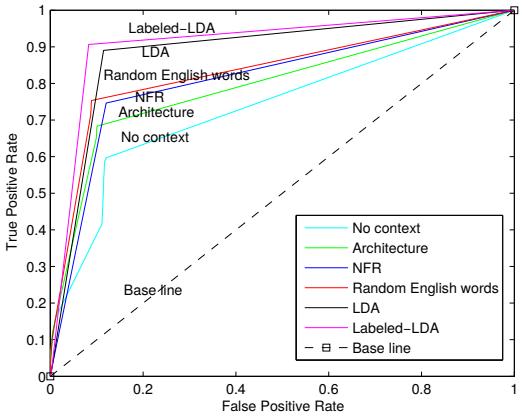


**Fig. 4** ROC curves of results from applying K-NN algorithm on Android reports.

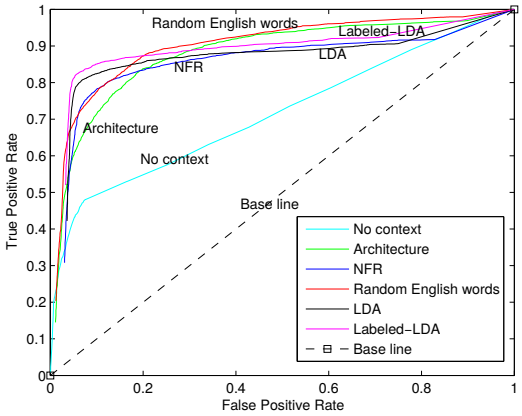


**Fig. 5** ROC curves of results from applying C4.5 algorithm on Android reports.

be used to recognize duplicate bug reports. But, our experiments demonstrate that there is a notable difference between the performance of our methods in the Android and the other bug repositories. We believe that this difference is due to an important difference in the structure of these bug repositories. As indicated in Figure 1, there is a considerably large bucket of duplicate reports in the Android bug repository (including 188 duplicate reports). We believe that the pattern of association between contextual features of the duplicate bug reports in the same bucket is what the machine-learners are leveraging to improve performance. Also, based on the fact that larger buckets provide more pairs marked as "dup", they provide more training data for the classifiers. Consequently, duplicate bug reports pertaining to the large buckets are easily

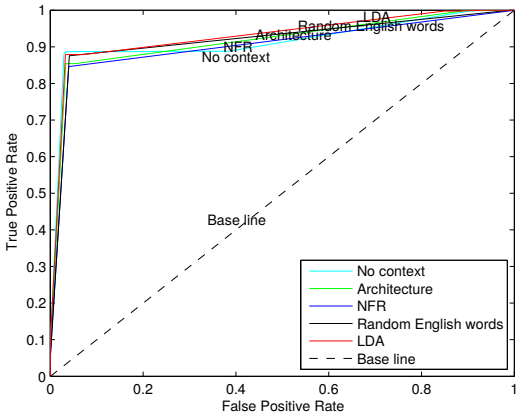


**Fig. 6** ROC curves of results from applying K-NN algorithm on Eclipse reports.

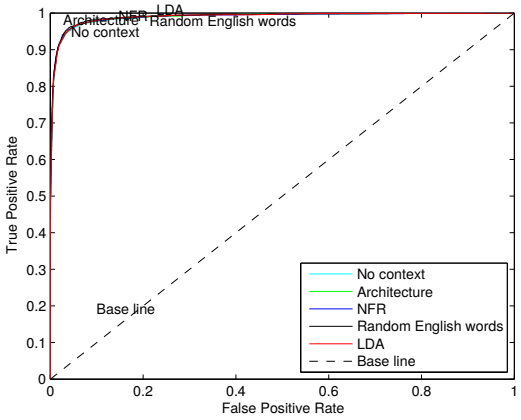


**Fig. 7** ROC curves of results from applying logistic regression algorithm on Eclipse reports.

recognized by the machine learners when they are provided with contextual features.

To examine this idea, we removed all the bug reports belonging to the largest bucket from Android bug repository and created a repository called Android_modified. Then, we conducted an experiment to trace the predictions made by the machine learners for the data-sets including only the LDA contextual features. In this experiment, the machine learner with the highest prediction performance is taken into consideration for each bug repository. We investigated some of the false negative predictions, *i.e.,* the "dup" instances not recognized by the machine learner. Also, we divided the buckets of bug reports into two groups: the large buckets (buckets including 10 or more duplicate bug reports) and the small buckets (the buckets including

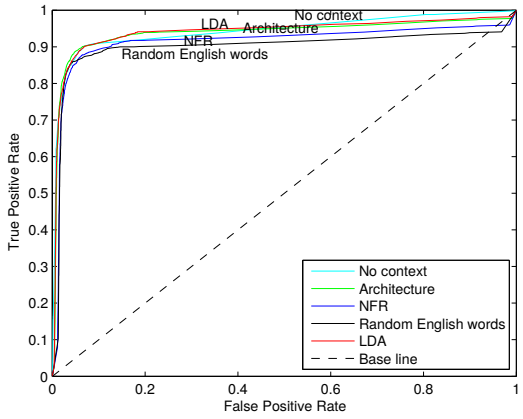


**Fig. 8** ROC curves of results from applying C4.5 algorithm on Mozilla reports.

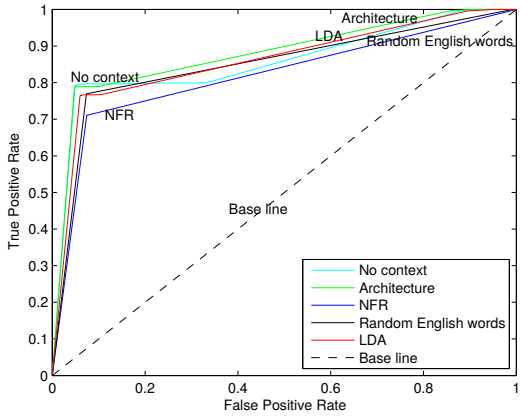


**Fig. 9** ROC curves of results from applying K-NN algorithm on Mozilla reports.

less than 10 duplicate reports). We found that 90% of the false negatives for the android_modified are from the small buckets while only 37% of all the pairs marked as "dup" are from these buckets. For the Eclipse repository, 88% of the false negatives belong to the small buckets while 75% of the "dup"s are from these buckets. The same experiment on the Mozilla repository indicated that 81% of the false negatives belong to small buckets while only 53% of the "dup"s are related to these buckets. Finally, for the OpenOffice bug repository, 82% of the false negatives belong to the small buckets while 68% of the "dup" instances are from these buckets. These results constitute evidence that machine learners can identify the duplicate reports belonging to the larger buckets more effectively in comparison to the duplicate reports from small buckets when applying contextual features. As a result, we realized

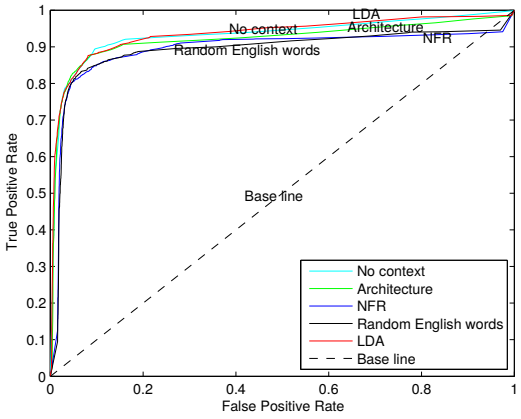


**Fig. 10** ROC curves of results from applying C4.5 algorithm on OpenOffice reports.

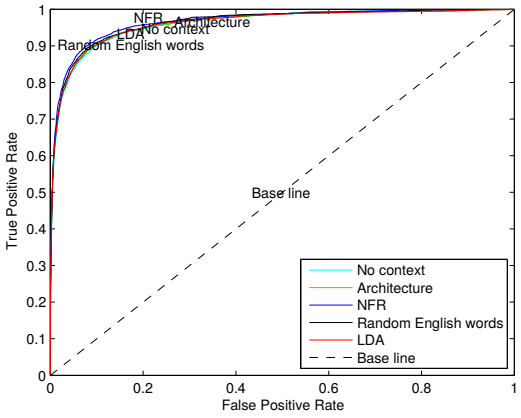


**Fig. 11** ROC curves of results from applying logistic regression algorithm on OpenOffice reports.

that classifiers identify the duplicate records belonging to large buckets more effectively than the duplicates from small buckets (like buckets of size 2 that are very common as demonstrated in Figure 1).

### 5.3 Effectiveness of Number of Features

As mentioned previously, each contextual data-set adds some new contextual features to each bug report. The 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) to the bug-deduplication process. Here we answer the following research question:

*Does adding more features (even junk) to the bug reports improve the accuracy of duplicate bug report detection regardless of their context?*

Figures 12, 14, 16, and 18 show the relationship between the kappa measure and the number of added features to Android, Eclipse, Mozilla, and OpenOffice bug reports respectively. Each box-plot in these figures 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 Figure 12, there is a little difference between the performance of Random English Word context and NFR context, but NFR adds 20 fewer features. Consequently, context is more important than feature count. Figures 14, 16, and 18 imply that although the English Random Words includes the maximum number of features, it resulted in the weakest performance among the other contexts. This result reveals that the number of added features is not effective in improving the detection of duplicate bug reports. And, it is indeed the context that impacts the prediction of duplicates.

The correlations between the number of added features and AUC are displayed in Figures 13, 15, 17, and 19. The AUC measure for Naive Bayes, Logistic Regression, K-NN, and C4.5 is demonstrated in these figures. Figure 13 shows the relation between the number of added features and AUC by fitting a linear regression function (the slope of this line is 0.0012). The measured correlation value for this figure is 0.46 which does not represent a high positive correlation. For Figures 15, 17, and 19 the slopes are $-0.0002$, $-0.0006$, and $-0.0008$ respectively which imply a small correlation between the number of added features and the efficiency of detecting duplicate reports. Taking into account the points mentioned above, it is evident that simply adding more features can not improve the performance of duplicate bug report detection.

## 5.4 Evaluating the List of Candidates

In this Section, we discuss the impact of context on filtering the bug reports and providing a list of candidate duplicate reports to triagers, based on the reported results of the MAP measure. This bug-report retrieval approach is different from the method applied in our recent work [2], in which the machine-learning classifiers are used to decide whether two bug reports are duplicates or not. Here, we aim to address the following research question: *Could the software context improve the quality of the list of candidate duplicates to an incoming bug report?*

Tables 16, 17, 18, and 19 report the MAP results for the Android, Eclipse, Mozilla, and OpenOffice bug reports respectively. In each one of these tables, three different types of similarity criterion functions are assessed exploiting the MAP measure. As addressed earlier in Section 4.5.2, these criteria are cosine similarity based, Euclidean distance based, and logistic regression based metrics.

Some of the conducted experiments exclusively make use of specific contextual information to retrieve the duplicate reports. Some other ones exploit

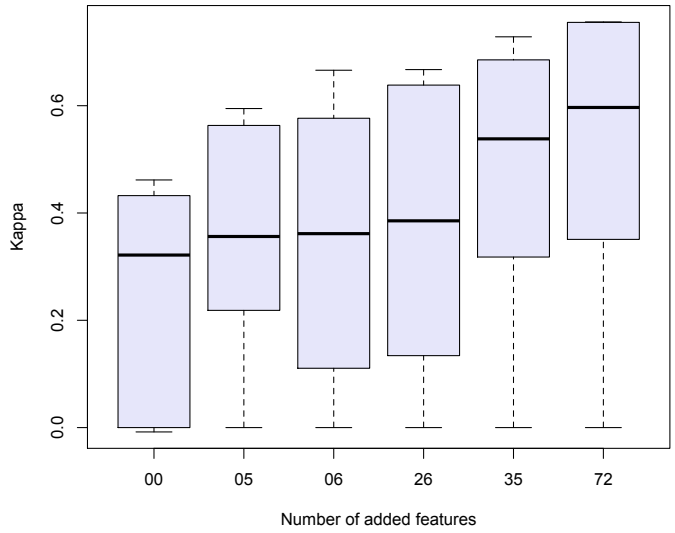


**Fig. 12** Kappa versus number of added features for the Android bug repository. The x axis shows the number of features each context adds to the bug reports (which is equal to the number of word lists of the contextual data). The contexts from left to right are no context, architecture, NFR, Random words, LDA, and Labeled-LDA.

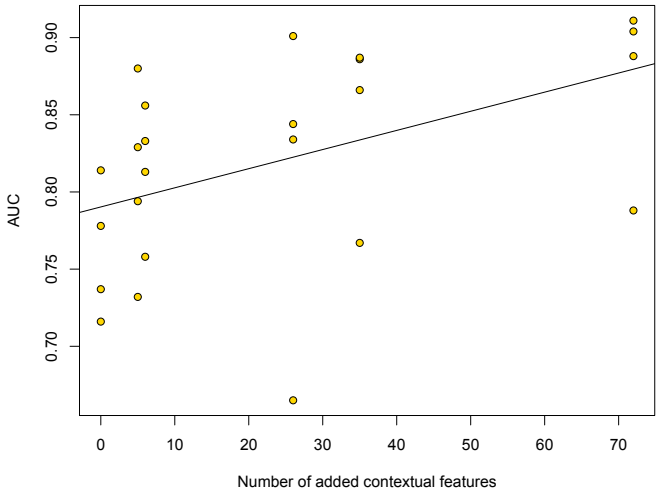


**Fig. 13** AUC versus number of added features for the Android bug repository. The x axis shows the number of features each context adds to the bug reports. The contexts from left to right are no context, architecture, NFR, Random words, LDA, and Labeled-LDA.

both the REP function and the contextual information. Moreover, in some of

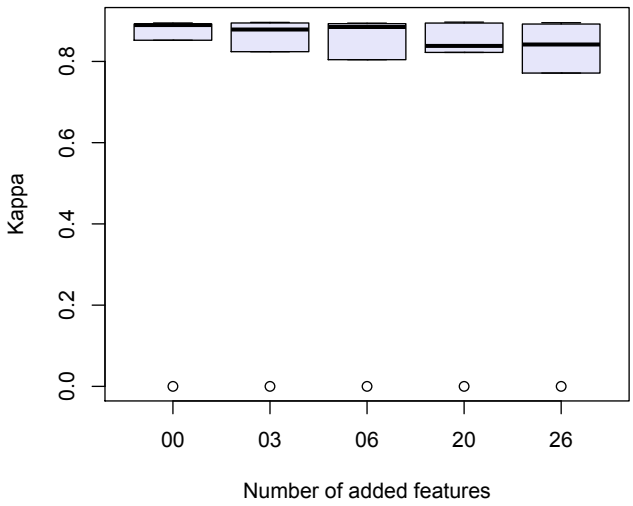


**Fig. 14** Kappa versus number of added features for the Eclipse bug repository. The x axis shows the number of features each context adds to the bug reports (which is equal to the number of word lists of the contextual data). The contexts from left to right are no context, architecture, NFR, Random words, and LDA.

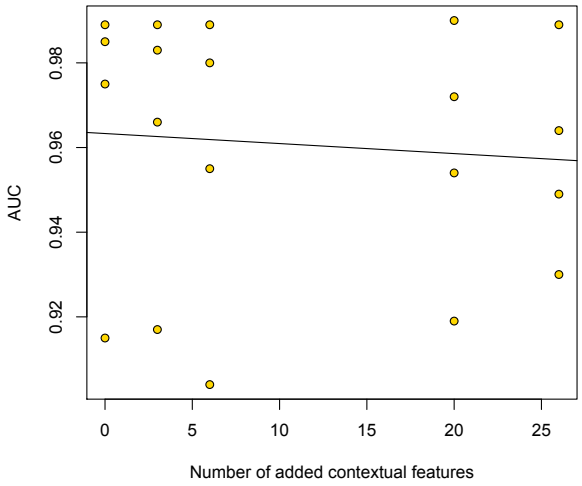


**Fig. 15** AUC versus number of added features for the Eclipse bug repository. The x axis shows the number of the features each context adds to the bug reports. The contexts from left to right are no context, architecture, NFR, Random words, and LDA.

the experiments, all the contextual information (except the Random English

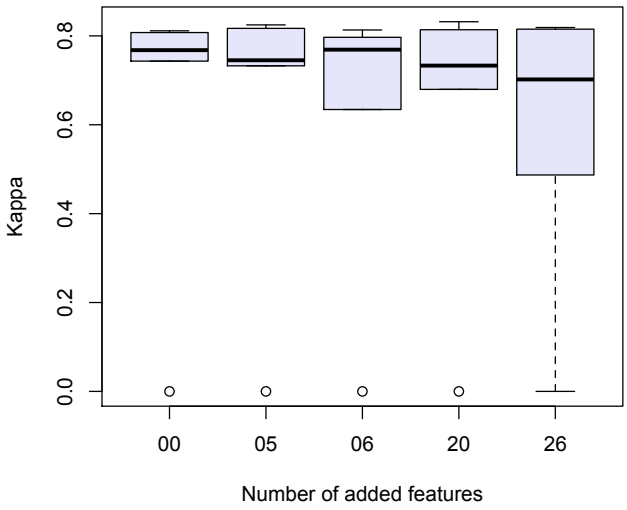


**Fig. 16** Kappa versus number of added features for the Mozilla bug repository. The x axis shows the number of features each context adds to the bug reports (which is equal to the number of word lists of the contextual data). The contexts from left to right are no context, architecture, NFR, Random words, and LDA.

words one) are applied without utilizing the REP function (all without REP) and with exploiting the REP function (all). The highest MAP value achieved for each repository is presented in bold in the tables.

As indicated in Tables 16, 17, 18, and 19, applying the logistic-regression based technique could make considerable enhancement in identifying duplicate reports. For instance, when the similarity criterion exclusively exploits the REP function, the logistic regression based approach provides helpful coefficients that boost the MAP value 9.5%, 7.8%, 9.1%, and 8.3% for the Android, Eclipse, Mozilla, and OpenOffice bug repositories respectively in comparison to the case of normal REP being applied as the similarity criterion. The boost in REP via logistic regression is probably similar to tuning REP as prescribed by Sun *et al.* [30]. Regardless in the case of MAP context's benefit is not always clear, but the re-weighting of REP with a logistic regression function is clearly beneficial as it outperforms plain REP in every case.

## 5.5 Discussion of Context

According to the experiments we discussed above, the added contextual data did not improve the duplicate report retrieval performance significantly and consistently. As reported in the tables, two of the repositories (Android and OpenOffice) showed that the combination of REP and contextual-similarity

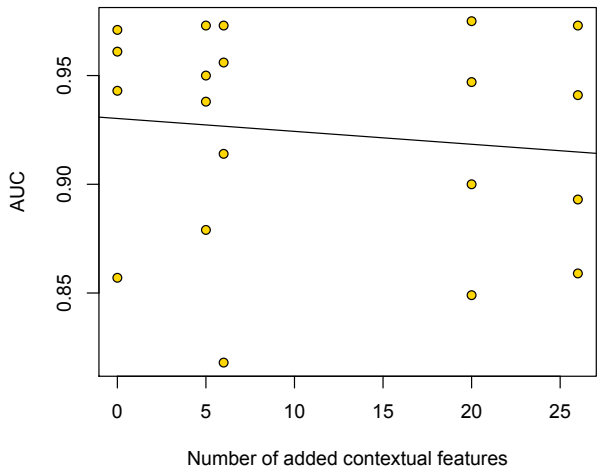


**Fig. 17** AUC versus number of added features for the Mozilla bug repository. The x axis shows the number of the features each context adds to the bug reports. The contexts from left to right are no context, architecture, NFR, Random words, and LDA.

measure is able to improve the performance of duplicate bug-report detection by up to 0.7%. However, two other repositories (Mozilla and Eclipse) did not show any improvement after applying the software context. In these cases REP did not do better than REP weighted by logistic regression. This implies that there is much potential for tuning. Consequently, we could not elevate the quality of the list of candidate duplicates greatly by considering the contextual data for some of the projects.

The value of the contextual approach is that a small amount of effort can often improve results or be combined with existing IR results. In Section 3.1 many of the architectural datasets took 2 person-hours or less to construct. The most expensive data-set was the fully-supervised labelled-LDA dataset taking 60 person-hours. Excluding labelled LDA, the effort is negligible. Often this effort is only a one-time cost that can be reused across potentially multiple systems.

When building contexts it seems one should try to be as close to the domain of the project as possible to increase the possibility of relevant matches. Sparse contexts can lead to sparse features that are not as reliable for deduplication. If features are rare or often missing, learners might ignore those features.

Features whose 3rd quartile threshold was at 0.0, that is features who are 0 for more than 75% of bug reports tended not to do perform well, except for LDA. Thus our recommendation is that a balance should be struck against the number of features and the size of the contexts. Contexts that do not intersect with the available bug reports will not be valuable features. The

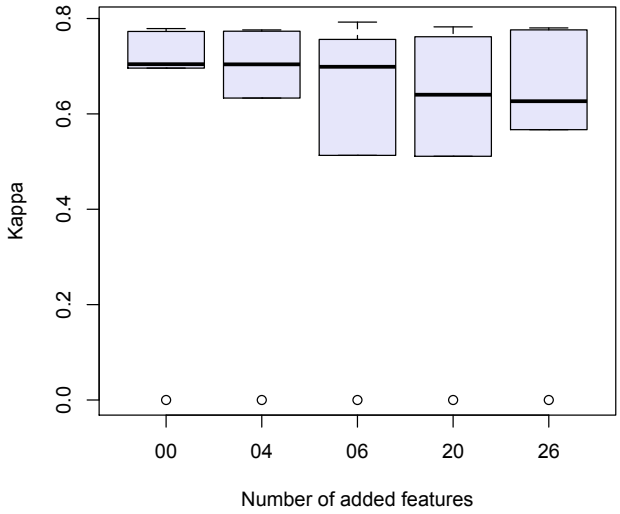


**Fig. 18** Kappa versus number of added features for the OpenOffice bug repository. The x axis shows the number of features each context adds to the bug reports (which is equal to the number of word lists of the contextual data). The contexts from left to right are no context, architecture, NFR, Random words, and LDA.

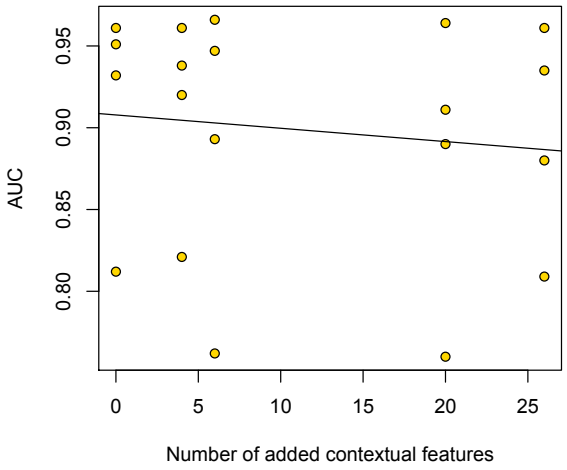


**Fig. 19** AUC versus number of added features for the OpenOffice bug repository. The x axis shows the number of the features each context adds to the bug reports. The contexts from left to right are no context, architecture, NFR, Random words, and LDA.

intent of contexts is to allow learners and functions to observe similarity and discriminate between differences.

**Table 16** MAP results for the list of candidates of Android bug repository

| Criterion | MAP |
|---|---|
| REP | 0.410 |
| Architecture context | 0.020 |
| NFR context | 0.200 |
| LDA context | 0.293 |
| LabeledLDA | 0.298 |
| REP and Architecture context (cosine similarity) | 0.376 |
| REP and NFR context (cosine similarity) | 0.301 |
| REP and LDA context (cosine similarity) | 0.324 |
| REP and LabeledLDA context (cosine similarity) | 0.330 |
| REP and Architecture context (Euclidean distance similarity) | 0.383 |
| REP and NFR context (Euclidean distance similarity) | 0.414 |
| REP and LDA context (Euclidean distance similarity) | 0.318 |
| REP and LabeledLDA context (Euclidean distance similarity) | 0.412 |
| REP (Logistic regression based) | 0.505 |
| all (Logistic regression based) | 0.459 |
| all without REP (Logistic regression based) | 0.097 |
| REP and architecture cosine (Logistic regression based) | 0.499 |
| REP and LDA cosine (Logistic regression based) | **0.513** |
| REP and Random English words cosine (Logistic regression based) | 0.479 |
| REP and NFR cosine (Logistic regression based) | 0.468 |
| REP and Labeled-LDA cosine (Logistic regression based) | 0.501 |
| Architecture context (Logistic regression based) | 0.042 |
| LDA context (Logistic regression based) | 0.365 |
| Random English words context (Logistic regression based) | 0.101 |
| NFR context (Logistic regression based) | 0.003 |
| Labeled-LDA context (Logistic regression based) | 0.374 |

## 5.6 Threats to Validity

Construct validity is threatened by our word-lists, their construction process and the degree to which their content actually represents "context" or just important tokens. Our measurements rely on the status of bug reports in some real-world bug-tracking systems that have a huge number of bug reports not processed by the triagers (have the status value of "New" or "Unconfirmed"). And, there may be many duplicate bug reports among them. Also, for the Android, Eclipse, Mozilla, and OpenOffice bug repositories we exploited in this study, there are only 2%, 6%, 8%, and 9% of the bug reports marked as "duplicate". 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. K-NN might not be appropriate to apply as the feature vectors are not necessarily comparable as distances.

External validity is threatened by the fact that some particular characteristics of a bug repository might lead to our experimental results. To reduce

**Table 17** MAP results for the list of candidates of Eclipse bug repository

| Criterion | MAP |
|---|---|
| REP | 0.379 |
| Architecture context | 0.008 |
| NFR context | 0.069 |
| LDA context | 0.072 |
| REP and Architecture context (cosine similarity) | 0.306 |
| REP and NFR context (cosine similarity) | 0.280 |
| REP and LDA context (cosine similarity) | 0.209 |
| REP and Architecture context (Euclidean distance similarity) | 0.367 |
| REP and NFR context (Euclidean distance similarity) | 0.372 |
| REP and LDA context (Euclidean distance similarity) | 0.245 |
| REP (Logistic regression based) | **0.457** |
| all (Logistic regression based) | 0.453 |
| all without REP (Logistic regression based) | 0.135 |
| REP and architecture cosine (Logistic regression based) | 0.455 |
| REP and LDA cosine (Logistic regression based) | 0.455 |
| REP and Random English words cosine (Logistic regression based) | 0.455 |
| REP and NFR cosine (Logistic regression based) | 0.456 |
| Architecture context (Logistic regression based) | 0.021 |
| LDA context (Logistic regression based) | 0.095 |
| Random English words context (Logistic regression based) | 0.037 |
| NFR context (Logistic regression based) | 0.077 |

**Table 18** MAP results for the list of candidates of Mozilla bug repository

| Criterion | MAP |
|---|---|
| REP | 0.208 |
| Architecture context | 0.005 |
| NFR context | 0.008 |
| LDA context | 0.018 |
| REP and Architecture context (cosine similarity) | 0.208 |
| REP and NFR context (cosine similarity) | 0.207 |
| REP and LDA context (cosine similarity) | 0.208 |
| REP and Architecture context (Euclidean distance similarity) | 0.169 |
| REP and NFR context (Euclidean distance similarity) | 0.203 |
| REP and LDA context (Euclidean distance similarity) | 0.089 |
| REP (Logistic regression based) | **0.299** |
| all (Logistic regression based) | 0.294 |
| all without REP (Logistic regression based) | 0.042 |
| REP and architecture cosine (Logistic regression based) | **0.299** |
| REP and LDA cosine (Logistic regression based) | 0.296 |
| REP and Random English words cosine (Logistic regression based) | **0.299** |
| REP and NFR cosine (Logistic regression based) | **0.299** |
| Architecture context (Logistic regression based) | 0.013 |
| LDA context (Logistic regression based) | 0.029 |
| Random English words context (Logistic regression based) | 0.015 |
| NFR context (Logistic regression based) | 0.013 |

this risk, we have used four large bug repositories related to different software projects in our experiments.

**Table 19** MAP results for the list of candidates of OpenOffice bug repository

| Criterion | MAP |
|---|---|
| REP | 0.238 |
| Architecture context | 0.003 |
| NFR context | 0.041 |
| LDA context | 0.038 |
| REP and Architecture context (cosine similarity) | 0.180 |
| REP and NFR context (cosine similarity) | 0.136 |
| REP and LDA context (cosine similarity) | 0.105 |
| REP and Architecture context (Euclidean distance similarity) | 0.234 |
| REP and NFR context (Euclidean distance similarity) | 0.236 |
| REP and LDA context (Euclidean distance similarity) | 0.237 |
| REP (Logistic regression based) | 0.321 |
| all (Logistic regression based) | **0.322** |
| all without REP (Logistic regression based) | 0.078 |
| REP and architecture cosine (Logistic regression based) | 0.321 |
| REP and LDA cosine (Logistic regression based) | 0.321 |
| REP and Random English words cosine (Logistic regression based) | 0.318 |
| REP and NFR cosine (Logistic regression based) | **0.322** |
| Architecture context (Logistic regression based) | 0.010 |
| LDA context (Logistic regression based) | 0.051 |
| Random English words context (Logistic regression based) | 0.021 |
| NFR context (Logistic regression based) | 0.052 |

## 6 Conclusions and Future Work

In this study, we have taken advantage of software contexts in addition to the textual and categorical similarity measurements to address the ambiguity of synonymous software-related words within bug reports written by users, who have different vocabularies. We assume that bug reports are likely to refer to a non-functional requirement or some functionalities related to some architectural components in the system. Thus, we have exploited the software contexts of software non-functional requirements, software architecture, and software topics extracted by LDA/Labeled LDA.

We replicated Sun *et al.*'s [30] method of textual and categorical comparison and extended it by adding contextual comparison, through the addition of contextual features to the bug reports exploiting the above mentioned software contexts. These features are taken into consideration in addition to the basic properties (description, title, type, component, version, priority, and product) of the bug reports while comparing the reports and measuring their similarities. We have conducted our experiments on the Android, Eclipse, Mozilla, and OpenOffice bug repositories. As a result, this contextual approach improved the accuracy of bug report deduplication by 0.1%-11.5% over Sun *et al.*'s [30] method.

Furthermore, we improved the quality of the list of candidate duplicates for an incoming bug report by applying the REP function [30], our contextual measures, and the logistic-regression classifier's probabilistic model. Consequently, we achieved 7.8%-9.5% improvement in Mean Average Precision (MAP) mea-

sure over Sun *et al.*'s [30] approach. Adding software context resulted in an improvement of the quality of the list of candidate duplicates for Android and OpenOffice bug repositories by up to 0.7%, but it did not improve the candidate-duplicate lists for the Eclipse and Mozilla repositories.

Comparing our method against the work of Sun *et al.* is possible because our experiment methodologies are quite "parallel". Unfortunately, however, in the absence of openly shared data sets and algorithm implementations an accurate comparison of all relevant duplicate-bug detection and retrieval algorithms is not practical. Nevertheless, our method favorably compares to the reported performance of most related research studies, as it improves on REP by adding context and with logistic REP.

Our experiments demonstrate that adding software contextual features to the bug reports can improve the performance of bug-report deduplication while retrieving the duplicates by the machine-learning classifiers. We feel that contextual features disambiguate bug reports and thus by adding the context, the classifiers can decide more efficiently if two bug reports are duplicates or not. On the other hand, adding the contextual features could not enhance the quality of the list of candidate duplicates for the majority of software projects.

This document describes one scenario where context matters: bug-deduplication. The value of software-development context was also demonstrated in prior work [16]. This study provides more evidence that we can achieve gains in bug-deduplication performance by including contextual features, prior information, into our software engineering related IR tasks, whether it is bug deduplication or LDA topic labelling and tagging. We hope this work motivates researchers to build more corpora of software concepts in order to improve automated and semi-automated software engineering tasks.

In the future, we plan to implement our method as an embedded tool in an issue-tracker to empirically investigate the role that this method can actually play in assisting the triagers and save their time and effort when looking for the duplicates of an incoming bug report. Future plans include evaluating what makes a good context. This way, we can take advantage of their helpful feedback to enhance the effectiveness and usability of our approach. As we have shown that context matters, we suspect that more modern, state-of-the-art bug deduplication techniques such as those by Nguyen *et al.* [25] can be further improved by integrating contextual features.

## 7 Acknowledgments

We would like to thank Sun et al. [30] for sharing their Eclipse, OpenOffice, and Mozilla datasets with us. Abram Hindle and Eleni Stroulia were supported by NSERC Discovery Grants.

## References


1. Karan Aggarwal, Tanner Rutgers, Finbarr Timbers, Abram Hindle, Russ Greiner, and Eleni Stroulia. Detecting duplicate bug reports with software engineering domain knowledge. In Yann-Gaël Guéhéneuc, Bram Adams, and Alexander Serebrenik, editors, *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 211–220. IEEE, 2015.
2. 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*, pages 183–192. IEEE Press, 2013.
3. J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39. ACM, 2005.
4. J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM, 2006.
5. N. Ayewah and W. Pugh. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 241–252. ACM, 2010.
6. 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*, pages 337–345. IEEE, 2008.
7. Amy Brown and Greg Wilson. *The Architecture Of Open Source Applications*. lulu.com, June 2011.
8. Chris Buckley and Ellen M. Voorhees. Evaluating evaluation measure stability. In *Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '00, pages 33–40, New York, NY, USA, 2000. ACM.
9. Android Community. Android Technical Information. `http://source.android.com/tech/security/`, 2013.
10. NeilA. Ernst and John Mylopoulos. On the perception of software quality requirements during the project lifecycle. In Roel Wieringa and Anne Persson, editors, *Requirements Engineering: Foundation for Software Quality*, volume 6182 of *Lecture Notes in Computer Science*, pages 143–157. Springer Berlin Heidelberg, 2010.
11. Alan Grosskurth and Michael W Godfrey. Architecture and evolution of the modern web browser. *Preprint submitted to Elsevier Science*, 2006.
12. 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*, pages 124–127. IEEE, 2012.
13. 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*, pages 83–92. IEEE, 2012.
14. S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301. ACM, 2002.
15. L. Hiew. *Assisted detection of duplicate bug reports.* PhD thesis, The University Of British Columbia, 2006.
16. A. Hindle, N. A. Ernst, M. W. 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*, pages 163–172. ACM, 2011.
17. 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*, pages 357–361. IEEE, 1994.
18. 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*, pages 52–61. IEEE, 2008.
19. Ahmad Kayed, Nael Hirzalla, Ahmad A Samhan, and Mohammed Alfayoumi. Towards an ontology for software product quality attributes. In *Internet and Web Applications and Services, 2009. ICIW'09. Fourth International Conference on*, pages 200–204. IEEE, 2009.
20. J. Langford, L. Li, and A. Strehl. Vowpal wabbit online learning project, 2007.
21. Sun Microsystems. The openoffice.org source project: Technical overview. `http://www.immagic.com/eLibrary/ARCHIVES/GENERAL/SUN/OPENOFCT.pdf`, 2000.
22. Maria Carolina Monard and Gustavo EAPA Batista. Learning with skewed class distrihutions. *Advances in Logic, Artificial Intelligence, and Robotics: LAPTEC 2002*, 85:173, 2002.
23. N. K. 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*, pages 202–207. ACM, 2009.
24. T. Nakashima, M. Oyama, H. Hisada, and N. Ishii. Analysis of software bug causes and its prevention. *Information and Software Technology*, 41(15):1059–1068, 1999.
25. A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 70–79. ACM, 2012.
26. D. Ramage, D. Hall, R. Nallapati, and C. D. 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*, pages 248–256. Association for

Computational Linguistics, 2009.
27. Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, Mike Gatford, et al. Okapi at trec-3. *NIST SPECIAL PUBLICATION SP*, pages 109–109, 1995.
28. 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*, pages 499–510. IEEE, 2007.
29. N. Serrano and I. Ciordia. Bugzilla, ITracker, and other bug trackers. *Software, IEEE*, 22(2):11–13, 2005.
30. 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*, pages 253–262. IEEE Computer Society, 2011.
31. 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*, pages 45–54. ACM, 2010.
32. A. Sureka and P. Jalote. Detecting duplicate bug report using character n-gram-based features. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 366–374. IEEE, 2010.
33. Jason Van Hulse, Taghi M Khoshgoftaar, and Amri Napolitano. Experimental perspectives on learning from imbalanced data. In *Proceedings of the 24th international conference on Machine learning*, pages 935–942. ACM, 2007.
34. Byron C Wallace and Issa J Dahabreh. Class probability estimates are unreliable for imbalanced data (and how to fix them). In *ICDM*, pages 695–704, 2012.
35. Byron C Wallace, Kevin Small, Carla E Brodley, and Thomas A Trikalinos. Class imbalance, redux. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 754–763. IEEE, 2011.
36. 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*, pages 461–470. ACM, 2008.
37. Hugo Zaragoza, Nick Craswell, Michael J Taylor, Suchi Saria, and Stephen E Robertson. Microsoft cambridge at trec 13: Web and hard tracks. In *TREC*, volume 4, pages 1–1. Citeseer, 2004.