Empirical Software Engineering
https://doi.org/10.1007/s10664-018-9643-4

Preventing duplicate bug reports by continuously querying bug reports

Abram Hindle[1] · Curtis Onuczko[2]

Published online: 20 August 2018

Abstract

Bug deduplication or duplicate bug report detection is a hot topic in software engineering information retrieval research, but it is often not deployed. Typically to de-duplicate bug reports developers rely upon the search capabilities of the bug report software they employ, such as Bugzilla, Jira, or Github Issues. These search capabilities range from simple SQL string search to IR-based word indexing methods employed by search engines. Yet too often these searches do very little to stop the creation of duplicate bug reports. Some bug trackers have more than 10% of their bug reports marked as duplicate. Perhaps these bug tracker search engines are not enough? In this paper we propose a method of attempting to prevent duplicate bug reports before they start: continuously querying. That is as the bug reporter types in their bug report their text is used to query the bug database to find duplicate or related bug reports. This continuously querying bug reports allows the reporter to be alerted to duplicate bug reports as they report the bug, rather than formulating queries to find the duplicate bug report. Thus this work ushers in a new way of evaluating bug report deduplication techniques, as well as a new kind of bug deduplication task. We show that simple IR measures can address this problem but also that further research is needed to refine this novel process that is integrate-able into modern bug report systems.

Keywords Duplicate bug reports · Issue reports · Deduplication · Information retrieval · Just in time · Continuously querying bug reports · Continuous query

Communicated by: David Lo

✉ Abram Hindle
hindle1@ualberta.ca

Curtis Onuczko
curtiso@bioware.com

[1] Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada

[2] BioWare ULC, Edmonton, Alberta, Canada

1 Introduction

When software stops working the end-user is often asked to report on their experience and describe the problem they encountered to the software developers. These reports become issue tickets or bug reports and are stored in issue trackers and bug trackers. If two users, whether they are QA, developers, testers, or end-users, experience the same problem and report it, their bug reports are referred to as duplicate bug reports. This happens frequently for numerous reasons: users do not search for duplicate reports, users do not understand duplicate reports, users could not find the duplicate report, etc. (Bettenburg et al. 2008; Rakha et al. 2015) End-users do not necessarily share the same mindset, terminology, architectural expertise, software expertise, and vocabulary as the developers of a product. Nor do end-users share the same knowledge of software development and the common software development processes. Thus duplicate bugs happen and they happen frequently.

Duplicate bug reports are a problem for developers, triagers, and QA personnel because they induce more work (Rakha et al. 2015; Bettenburg et al. 2008). If a developer addresses 2 bug reports that report the same issue, they might find that they have wasted their own time searching for a bug that has already been fixed. Triagers have the same problem, they often must look for duplicate reports in an effort to close or link related bug reports.

Most users try to prevent duplicate bug reports by searching for existing bug reports first via keyword querying. This functionality exists in most bug trackers, such as Bugzilla, github issues, and JIRA. Search via keyword querying is very different than querying by example (Lukins et al. 2008). Most literature queries duplicate bug reports by example: they use the whole documents as their search query (Lukins et al. 2008; Wang et al. 2008; Sun et al. 2010; Alipour et al. 2013). Thus the method used for duplicate bug report detection is typically not the method that users employ.

Fundamentally, users search for duplicate bug reports before their bug report exists – once they make a duplicate bug report they have made a problem for triagers and developers.

Thus *as a research community are we addressing duplicate bug reports appropriately*? Shouldn't we be addressing how to prevent duplicate bug reports instead of how to cluster already existing duplicate bug reports? We argue in this paper that duplicate bug reports should be prevented before they start.

We approach preventing duplicate reports by recognizing that querying often fails. If the user already searched for duplicate bug reports and did not find them, they will start writing a bug report. Since current research already uses documents as queries why don't we use the document that the user is writing as the query. Furthermore as the user writes more words let us keep querying and trying to find duplicate bug reports based on the text of the bug report they have already started writing. Thus we want to stop duplicate bug reports, before they start, by using the inchoate or in-progress bug report as numerous queries against the bug tracker. We call this *Continuously Querying Bug Reports*, or *Continuously Querying* for short. This means that we provide suggestions of duplicate bug reports as the user is writing their bug report. This enables the user to stop at any time when they see a suggestion of a likely duplicate bug report. This solution is more query intensive as we have to query the bug tracker for every few words typed, but we show statistically that one can find duplicate bug reports quite quickly within the first 5 to 12 words of a bug report on average assuming a duplicate exists (see averages in Tables 2, 4, and 3).

A continuously querying implementation for an end-user could be a sidebar of suggested duplicate bug reports that pop up as the user types in their bug report. As each word typed in, the bug tracker is queried. If the user sees an interesting bug report they can interrupt inputting their current report and go investigate one of these suggestions to see if someone

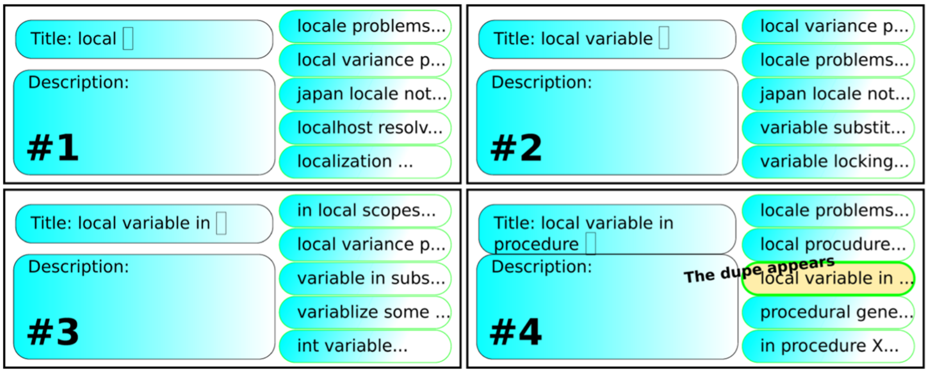


Fig. 1 Continuously Querying UI example whereby a user is typing in a bug report and the query results appear on the sidebar of the bug report input widget. The duplicate report that was queried is highlighted for the reader—the user interface would not this was the duplicate know but could use similarity scores to hide, show, or highlight certain results

has already reported their problem. Figure 1 provides a visual example. This means that continuously querying evaluation seeks duplicates sooner with fewer words than later.

In this paper:

- We propose *Continuously Querying Bug Reports* with *Continuously Querying*;
- We statistically evaluate the feasibility of the approach;
- We provide and share a large dataset to evaluate feasibility;
- We discuss and propose a variety of evaluation measures.

2 Prior work

Prior work relevant to continuously querying bug reports includes information retrieval, code recommenders, and studies and tools for duplicate bug report detection and bug report deduplication.

2.1 Information retrieval

Information Retrieval (IR) is the attempt to query for information, often unstructured text, from a repository of documents. The Google Suggest API (Google 2016) suggests auto-completions for search query terms. As we type in a search query it queries the Google

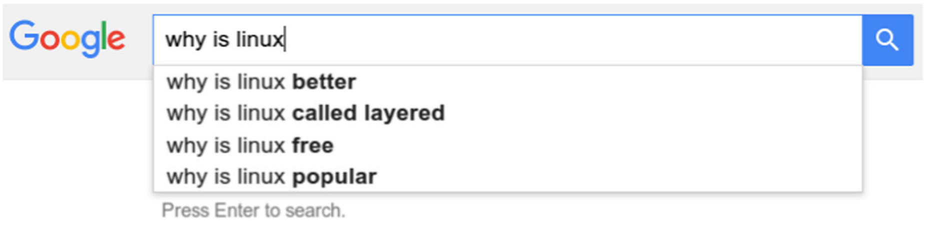


Fig. 2 Google Suggest suggesting possible query completions

Suggest API and provides possible query suggestions. Figure 2 depicts a screen-shot of Google Suggest API in action. Google suggest allows us to navigate to specialize queries that are more appropriate for our needs. The same can be done for bug reports.

Panichella et al. (2016) describe using genetic algorithms to optimize and tune the choice of IR techniques and models to fit the task at hand. They employ LSI for software traceability tasks but the general idea of parameterizing the IR model is quite relevant to this work.

A common model used in information retrieval is TF-IDF: term frequency multiplied by inverse document frequency. This produces a vector of normalized word counts by "interestingness" in terms of how rare a word is. Cosine distance is the angle distance between two vectors and is a common distance function used to compare documents because it normalizes word counts and vectors by size. In Panichella et al. (2016) they found many of their optimal configurations used the TF-IDF vector space model (VSM) and cosine distance. Much like TF-IDF and cosine distance, Sun et al. (2011) exploited BM25 and BM25F for query result scoring and ranking. VSM and different configurations of TF-IDF have been explored in bug localization by Wang et al. (2014) to find impacted files.

Continuously Querying is a naive kind of query reformulation as described by Haiduc (2014). Essentially the lack of query reformulation with in bug trackers makes Continuously Querying necessary. Query reformulation can be combined with Continuously Querying by trying to form a better query from the current in-progress bug report words.

2.2 Bug report deduplication

Bug report deduplication, also known as *duplicate bug report detection*, has been a hot-topic in software engineering research. Bug report deduplication is the querying of similar bug reports in order to cluster and group bug reports that report the same issue. Common tools in bug report deduplication are NLP (Runeson et al. 2007), machine-learning (Bettenburg et al. 2008; Sun et al. 2010; Alipour et al. 2013; Lazar et al. 2014), information retrieval (Sun et al. 2011; Sureka and Jalote 2010), topic analysis (Alipour 2013; Alipour et al. 2013; Klein et al. 2014), and deep learning (Deshmukh et al. 2017). Duplicates also appear in other software artifacts, as Zhang et al. (2015) have applied typical bug-deduplication technology to StackOverflow duplicate question detection. Sabor et al. (2017) discuss DURFEX and method of extracting features from bug reports with a focus on stack traces and reducing the vocabulary or number of features from a stack report. By smart extraction of features from stack traces and bug report fields they achieve good deduplication performance for bug reports with stack traces. Similar ideas of applying IR techniques been have to executed on stack traces from crash repositories (Campbell et al. 2016). Deshmukh et al. (2017) applied deep learning techniques to create a function that compares bug reports in order to find duplicate bug reports. They achieved good performance in terms of accuracy and recall.

Thung et al. (2014) have published about a bug report deduplication tool that provides duplicate bug report suggestions based on a vector space model. Rocha et al. (2015) discuss a similar tool. Rakha et al. (2015) discuss the effort it takes to address duplicate bug reports, firstly implying there is a cost to duplicate bug reports, but not all duplicates are equal, some are quite trivial.

The closest related work to this work is by Rakha et al. (2018), who discuss another kind of pre-submission bug deduplication used with modern Mozilla bugzilla trackers called *Just In Time* (JIT) duplicate retrieval. This differs from continuously querying in the sense that once the document is mostly written it is used to query for similar bug reports already committed. Rakha et al. (2017) also discuss similar issues of time of bug reports in evaluation and appropriate evaluation of duplicate bug reports.

Generally for the majority of these tools, studies, and cases the bug reports are assumed to have already been written and submitted to the bug tracker for a bug triager to triage.

2.3 Continuous query in database systems

In database systems research continuous query often refers to the property of streaming databases (Chandrasekaran and Franklin 2002) whereby a a set of queries is posed and the result set of each query continuously changes as new data is added to or modified in the database (Arasu et al. 2006; Chandrasekaran et al. 2003; Shah et al. 2003). Many of these system use terms such as real-time which can refer to hard real-time constraints or providing answers quickly (Kao and Garcia-Molina 1994). This is different than the proposed continuously querying method which queries the database of bug reports numerous times with different queries about the same problem as the document is typed in.

Continuously Querying Bug Reports could be viewed as a modification of *Real-Time Query Expansion* without the expansion part as the focus is on finding documents, not queries (White and Marchionini 2007).

We try to avoid confusing by calling our method *continuously querying* or *continuously querying bug reports* rather than continuous query.

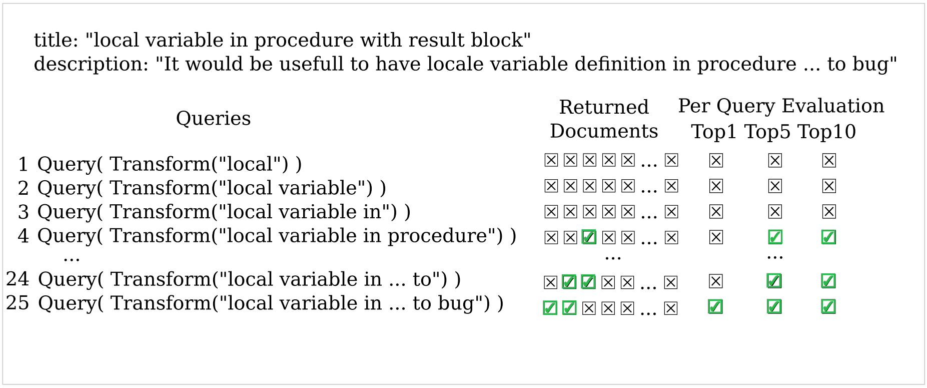
2.4 Code recommenders

Recommender systems are a relevant topics to continuously querying bug reports. Ponzanelli et al. (2013, 2014) describe systems that provide StackOverflow help as the user types in source-code, using the user's source code as the query. This is much like the proposed continuously querying except far more elaborate, as they do much query preprocessing for source code. Asaduzzaman et al. (2014) describe a system of contextual code completion which uses the current document as a query as well. The difference between these recommenders and continuously querying is more so the domain, continuously querying is initially envisioned for bug reports.

3 Continuously querying bug reports

Continuously Querying Bug Reports, or *Continuously Querying* for short, is the attempt to continuously query relevant documents during the creation of a document—as the document is being created. The intent of Continuously Querying Bug Reports is to alert users as they create a bug report that there are other potential duplicate bug reports already in existence. A user who notices their bug report is a duplicate should abandon their bug report and join the duplicate report found through Continuously Querying.

The process of *continuously querying bug reports* is demonstrated by example in Fig. 1 whereby as each new word is added to the bug report a new query is made. Figure 1 shows a prototypical example of a UI for a continuously querying implementation. It provides new suggestions based on each new term typed in. The query results are returned and displayed to the user near the bug report text entry widget. The user can then decide whether or not to investigate any of these potentially duplicate bug reports. Upon each word entry a new query is made and this list is updated. In Figs. 1 and 3 it takes until query 4 , "local variable in procedure", to find a duplicate bug report that is returned at rank 3. By the 24th and 25th word, in Fig. 3, 2 potential duplicates have shown up in the TOP 3 suggestions. Thus a continuously querying implementation for the user works as follows: For each word you



title: "local variable in procedure with result block"
description: "It would be usefull to have locale variable definition in procedure ... to bug"

	Queries	Returned Documents	Per Query Evaluation Top1	Top5	Top10
1	Query(Transform("local"))	☒ ☒ ☒ ☒ ☒ ... ☒	☒	☒	☒
2	Query(Transform("local variable"))	☒ ☒ ☒ ☒ ☒ ... ☒	☒	☒	☒
3	Query(Transform("local variable in"))	☒ ☒ ☒ ☒ ☒ ... ☒	☒	☒	☒
4	Query(Transform("local variable in procedure"))	☒ ☒ ✓ ☒ ☒ ... ☒	☒	✓	✓
...		...		...	
24	Query(Transform("local variable in ... to"))	☒ ✓ ✓ ☒ ☒ ... ☒	☒	✓	✓
25	Query(Transform("local variable in ... to bug"))	✓ ✓ ☒ ☒ ☒ ... ☒	✓	✓	✓

Fig. 3 Example of Continuously Querying Queries and Continuously Querying Bug Reports on a single bug report. A bug report is sequentially, word by word, to produce ever larger queries intent on finding similar bug reports, as the user types in the bug report

type into a bug report, up until a threshold, the continuously querying implementation will query the bug tracker for bug reports similar to the bug report you are making, displaying the ranked results near your input.

This is different than the Google Suggest API (Google 2016), which is used to suggest search queries as the user types a search query. "Google Suggest" for bug trackers would be to complete common bug tracker search queries or perhaps to help complete text being typed into the bug report submission form. Thus the Google Suggest API provides suggestions for queries as you type, a continuously querying implementation evaluates your input text as *the query* as you type. Continuously Querying Implementation do not "code complete" or "word complete" your bug report, instead they uses your in-progress bug report as a query to find similar bug reports.

This Google Suggest method (Google 2016), employed by continuously querying, is also employed in software engineering works such as the StackOverflow Code Snippet querier by Ponzanelli et al. (2013, 2014) that continuously queries of StackOverflow for example source code snippets, and contextual code completion by Asaduzzaman et al. (2014) that uses the entire live document as a query for code completion.

Continuously Querying is different than classical IR document querying because it asks many queries for 1 document against a corpus. Therefore former measures of effectiveness such as *mean reciprocal rank* (MRR), *mean average precision* MAP, or even accuracy have trouble dealing with these numerous queries per document.

3.1 Continuously querying bug reports clarified

In this paper *Continuously Querying* refers to *Continuously Querying Bug Reports* while writing a bug report. *Continuously Querying Implementation* refers to implemented *continuously querying* systems, such as DüpeBuster, or our gensim continuously querying implementation. *Continuously Querying Algorithm* refers to a method that could be implemented in a *continuously querying implementation*. *Continuously Querying Query* refers to a single query in a set of queries that are generated continuously via a *Continuously Querying Queries* refers to the set of each *continuously querying query* necessary to search for a document. *Continuously Querying* is the act of using a *continuously querying implementation* to make *continuously querying queries*.

3.2 Evaluation methodology

How do we evaluate the effectiveness of continuously querying implementation? We type in a bug report and look to see how soon duplicate bug reports appear.

Expected evaluation takes a duplicate of a bug report and produces a series of queries containing more and more words from the duplicate bug report until a maximum threshold has been met. Given an example bug report of, "This is a bug report", queries of "This", "This is", "This is a", "This is a bug", "This is a bug report" will be made. Figure 3 provides an example of the progression of these queries. A good Continuously Querying Algorithm shows duplicate bug reports sooner in terms of time and rank.

There are three main contexts of evaluation: time agnostic validation; continuous training where bug reports are added one after another; and historical splits validation. Each of these evaluation schemes will train models with bug reports and test these models with duplicate bug reports, typed in 1 word at a time until a threshold number of words (25 words as explained in Section 3.3) is met.

The first context, time agnostic validation, is when one queries an entire collection of bug reports with an existing bug report. When the repository is queried, the bug report being queried is excluded from the results and its duplicates are sought. This context ignores time. The assumption is that a corpus exists and that the creation time of a document or its duplicates is irrelevant. This context is unrealistic but enables cross-fold validation and due to a lack of empirical data (duplicate bug reports) this often is the most pragmatic method of evaluation. Cross-folds also have the problem of not having relevant duplicate reports in the training/query set to find. This context has a fundamental threat to validity that by training on future bug reports we are relying upon time travel to build a model. This context is referred to as "classical evaluation" by Rakha et al. (2017). This context also is inaccurate, Rakha et al. (2017) claim this kind of evaluation boosts performance measures unrealistically by 17 to 42%. An example of time agnostic evaluation is the work of Deshmukh et al. (2017) where the vocabulary for the deep neural network is extracted from all of the bug reports across time, rather than from the training set itself.

The second context, continuous training, is that of evaluation in order of time. This context is quite realistic as it models the slow build up of a bug report repository. Each bug is committed and trained upon in order, and if that bug report is a duplicate report then it is used for evaluation immediately and then trained upon. Unfortunately to evaluate this model we need to train the model with the past up to the bug report being queried. This means for a system with 1000 duplicate bug reports we need to train 1000 models – compared with 10-fold cross validation, this is 100-times more models. This context is unfair to models that can have expensive retraining, such as those that employ *Latent Dirchlet Allocation* (LDA) or *Latent Semantic Indexing* (LSI).

The third context, historical split validation, chooses N pivot bug reports where the report and its past will form the training set while any report after than report will form the evaluation set. This is different than cross-fold validation because the training set size and evaluation set size will vary depending on the the bug report being used as the pivot. So the last pivot will have a large training set and a small test set, and the first pivot will have a large test set and a small training set. The pivots are meant to equally partition the sequence of bug reports. This can produce cases where there are 0 duplicate bug reports in the test set. This addresses the unrealistic time-traveling concerns of the time agnostic first context.

It is worse for realism than the second continuous training context, but it requires less models and training to produce an evaluation. Given $N = 100$ and 1000 duplicate bug reports, only 100 models have to be made rather than 1000 models in the continuous case. Since the splits are time-ordered there is no time traveling while training.

A forth context was implemented by Rakha et al. (2017) is like the third context except to take a constant N duplicate bug reports out to tune the model at that point. So instead of a train and test split, they employ a train partition, a tuning partition of constant N size, and a test partition.

In this paper *we use the third context*, historical split validation. The other contexts can be used for valid evaluations but they might limit reproducibility or applicability of the results. The first context's lack of time awareness is a potential threat to validity so the results might not be relevant to real world performance. The second context's continuous updating works fine for models that train quickly or are updated quickly but if they rely on a heavier technique that needs to evaluate all the bug reports again using $O(n)$ or worse algorithms, like LSI/LSA or LDA (Alipour et al. 2013; Klein et al. 2014; Nguyen et al. 2012), it might be too costly to evaluate as you have to re-train on each document. The fourth context from Rakha et al. (2017) is well designed and appropriate. We did not rely on tuning in this work, which is a potential weakness, but the fourth context does address it. It is potentially more realistic than the 3rd context because you only test so far into the future before retraining again. The third context is a compromise between the number of evaluations and the number of models one has to generate.

3.3 Evaluation measures

How can we evaluate the effectiveness of a continuously querying implementation or algorithm?

We argue that an effective continuously querying implementation will return the duplicate bug report to a query formulated of words from that bug report. The sooner the duplicate bug report appears, the better the performance of the continuously querying algorithm. A continuously querying algorithm that needs 25 words to find a duplicate report is not as effective as one that needs only 5 words.

The threshold of 25 words was chosen because it is 2-3 times the length of the title of a bug report and 1/5 to 1/10th the size of the description of the bug report (see Table1). Any more would require a lot more time for evaluation. If we look at Fig. 4 of TOP-1 and TOP-5 performance—how often does the duplicate appear as the first suggestions or top 5 suggestions —we can see that there are elbows or pivot points at 25 and 50 words for different projects. 100 words often leads to better query performance for many projects but it is 4X the number of queries to evaluate. Since 25 words is half of the queries necessary to evaluate than 50 yet provides similar or results we chose 25 words as the threshold.

A continuously querying algorithm can only return so many duplicate bug report suggestions at a time. As per search engine IR methodology more than 10 results is often too many. But these are a multitude of queries so only so much can be evaluated, for example Google Suggestions uses 4 results for 1 query. Many prior works in bug deduplication, localization, and triage use top 5 results (Bettenburg et al. 2008; Wang et al. 2008, 2014). Thus we argue that top 5 results are probably the maximum amount of readable context – yet a result that occurs earlier is even better. We argue that a good continuously querying algorithm will return relevant duplicate bug reports sooner, thus higher rank correct results are valued more than lower rank correct results. This implies that treating rank reciprocally or as a weighted factor is important to evaluation.

Table 1 Duplicate Bug Report Data Sets and the Papers that cite them

Project	Source	# Bug Reports	# Dupes	# Dupe Buckets	Mean Title	Mean Description	Earliest	Latest	Prior Works
Android	GoogleCode	37626	1363	788	10.447	182.789	2007-11-12	2012-09-19	(Aggarwal et al. 2015; Alipour et al. 2013; Klein et al. 2014)
App Inventor	GoogleCode	2098	265	140	8.343	102.246	2010-09-09	2012-05-24	
Bazaar	LaunchPad	7020	523	523	9.898	439.733	2005-09-16	2016-04-05	
Cyanogenmod	GoogleCode	5185	677	403	8.552	245.319	2009-08-18	2012-05-17	
Eclipse	BugZilla	45234	4341	3382	9.891	199.899	2008-01-01	2008-12-30	(Aggarwal et al. 2015; Alipour et al. 2013; Sun et al. 2011, 2010; Rakha et al. 2017)
K9Mail	GoogleCode	4309	909	471	8.428	205.426	2008-10-28	2012-05-24	
Mozilla	BugZilla	75648	10479	7050	11.233	188.412	2010-01-01	2010-12-31	(Aggarwal et al. 2015; Alipour et al. 2013; Sun et al. 2011, 2010; Wang et al. 2008; Jalbert and Weimer 2008; Rakha et al. 2015, 2017)
MyTrack	GoogleCode	921	126	80	8.793	147.263	2010-05-05	2012-05-15	
OpenOffice	BugZilla	31136	4460	2799	9.161	179.748	2008-01-02	2010-12-21	(Aggarwal et al. 2015; Alipour et al. 2013; Sun et al. 2011, 2010; Rakha et al. 2017)
OpenStack	LaunchPad	17077	211	211	10.326	331.659	2011-04-26	2016-04-06	
osmand	GoogleCode	1026	73	58	8.329	119.935	2010-04-26	2012-05-25	
Tempest	LaunchPad	2085	98	98	9.132	389.582	2011-09-09	2016-04-06	

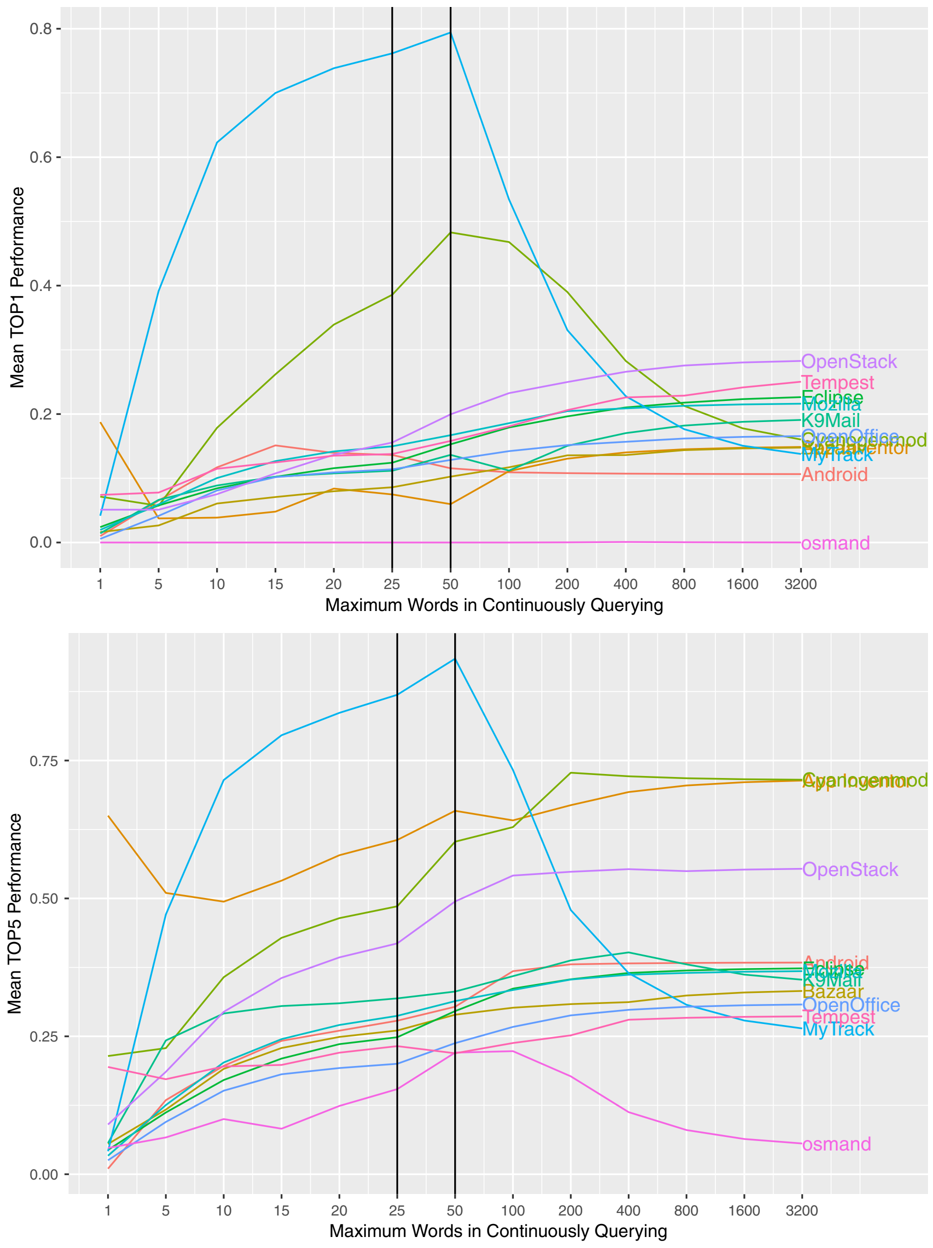


Fig. 4 TOP-1 and TOP-5 performance of Parameter sweep of word thresholds at 10 splits from 1 to 3200 words

All of the following evaluation scores give a score ranging from 0.0 to 1.0 where 1.0 is preferred or perfect and 0.0 is either no correct results, not found, or no matches. When averaged all of these scores have averages between 0.0 and 1.0. The queries will be duplicate bug reports, as we only have trust-able information on true-duplicate bug reports and have no cases of marked non-duplicate bug reports. Thus we assume that when bug reports are marked as duplicate they are indeed duplicate.

3.3.1 Classical IR and bug report deduplication evaluation measures

In this section we describe the measures TOPk, MAP, and MRR which are commonly used in IR and bug report deduplication literature (Sun et al. 2011; Bettenburg et al. 2008; Jalbert and Weimer 2008; Alipour et al. 2013; Aggarwal et al. 2015; Klein et al. 2014; Sureka and Jalote 2010; Runeson et al. 2007; Lazar et al. 2014).

TOPk: *TOPk* evaluation is where given $k \in 1, 5, 10$ a query is viewed successful if a duplicate bug report is found within the top k results. So if the first result is the duplicate bug report, then *TOP1*, *TOP5*, and *TOP10* all get a score of 1.0, if none or 0 of the to k results are relevant a score of 0 is given. Then the reported TOPk is average of these scores: the sum of scores divided by the number of queries. A TOP1 of 0.5 means that in half of the queries the first result was the duplicate bug report. When we report TOPk for a set of bug report we are reporting average TOPk. Thus for 10 bug reports each bug report has a TOPk score calculated and the average of these TOPk scores is reported for that group of bug reports.

$$in_k(i) = \begin{cases} 1 & \text{if } index_i \leq k \\ 0 & \text{otherwise} \end{cases}$$

$$TOP_k(Q) = \frac{1}{|Q|} \sum_{i=1}^{|Q|} in_k(i)$$

Where $index_i$ is the minimum ranked duplicate bug report.

MRR: TOPk does not emphasize the importance of the rank of the correct suggestion, it just uses a cut-off. MAP and MRR focus on the rank of the correct answer. *MRR*, *mean reciprocal rank*, gives us the average of the reciprocal rank of the correct answer in a query. This means that correct answers with low ranks are punished. Mean reciprocal rank only supports 1 correct answer—which is inaccurate for duplicate bug reports since there can be many possible duplicates in the same duplicate bug report cluster. Some prior work uses MRR for bug de-duplicating by suggesting only the first match matters (Sun et al. 2011). To evaluate continuously querying we calculate MRR per bug report, because each bug report consists of multiple queries. The mean of the reciprocal rank for the correct answer of the queries is used. If there is no correct answer 0 is scored. MRR is unrealistic because it evaluates results further down than a user would read. MRR does not have a cap rank like TOPk. For multiple bug reports we usually take the average of mean reciprocal rank for all bug reports (average mean reciprocal rank).

$$MRR(Q) = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{index_i}$$

MAP: MRR is convenient but unrealistic when there is more than 1 correct answer. With duplicate bug reports, the reports are part of a cluster so having multiple correct answers is important, as well having multiple correct answer early (high ranks such as 1 to 5). MAP or Mean Average Precision is an information retrieval score that ranks the effectiveness of a set of queries and punishes query results that are correct but lower ranked. This means that a result at rank 5 is scored far worse than a result at rank 2. MAP doesn't have a cap rank like

TOPk. Also MAP allows for multiple matches. MAP calculates aveP (average precision) of each query and then gives the mean of the aveP for the set of queries. In the case of continuously querying evaluation MAP is used to measure 1 bug report's set of continuously querying queries. When evaluated with multiple bug reports we will report the average MAP (average mean average precision[1]). MAP is more appropriate for bug deduplication because sometimes there is more than 1 duplicate bug that could be returned.

$$MAP = \frac{\sum_{q=1}^{|Q|} aveP(q)}{|Q|} \tag{1}$$

$$aveP(q) = \frac{\sum_{k=1}^{n} P(k) \times (Rel(k))}{number\_of\_relevant\_documents} \tag{2}$$

Where n is the number of returned documents and $P(k)$ is the precision at k. $Rel(k) = 1$ when the documents are duplicates of each other and $Rel(k) = 0$ otherwise; relevant documents are either all possible duplicates or those duplicates seen in the result set.

3.3.2 Proposed continuously querying evaluation measures

We feel that the prior classical IR measures do not describe Continuously Querying Algorithm performance well. Continuously Querying performance evaluation should consider how soon a result is received. Thus based on MAP and MRR we pose 3 metrics, AveP-TOP5, MRRTOP5, and MRRTOP5^{-1}, that attempt to give more weight to methods that return successful queries sooner. These measures assume that a very limited set of bug reports will be shown to the user (5), and they punish results which require more words (continuously querying queries), rather than less, to have a successful duplicate bug report appear.

AveP-TOP5: A measure that is arguably more relevant to continuously querying is *AveP-TOPk*. AveP-TOP5 helps determine if we find our duplicates quickly or not as those queries that need too many words (continuously querying queries) to find duplicate bug reports should be scored lower. AveP-TOPk is a combination of MAP and TOPk. AveP-TOPk first takes the continuously querying queries and ranks them by TOPk, thus each continuously querying query is evaluated as 1.0 or 0.0 depending if a duplicate result occurs in its top k results. Then the results are treated as a single query and used in average precision. Thus the earlier that a result appears in the top5 the higher the value AveP-TOPk value and the later it appears the lower the AveP-TOPk value. Thus this measurement boosts the performance of continuously querying algorithms who return duplicates in the fewest number of words. Imagine a set of continuously querying queries for 5 words for the document, "at the small dog house", where all 5 continuously querying queries reveal a duplicate in their top 5. AveP-TOP5 would result in 1.0 $((1/1 + 2/2 + 3/3 + 4/4 + 5/5)/5)$. If only the last 3 continuously querying queries ("at the small", "at the small dog", "at the small dog house") returned top5 the AveP-TOP5 score would be 0.478 $((1/3 + 2/4 + 3/5)/3)$, and if only the last query worked 0.2 $((1/5)/1)$. Thus AveP-TOPk promotes queries that result in good TOPk results sooner than later. It is essentially MAP and TOPk mixed together. Once we take the average or mean AveP-TOPk over many queries we effectively have something similar to MAP except that it is realistic in the number of query results that might be

[1]average mean average precision rolls off the tongue but perhaps triple mean precision sounds better.

evaluated. When AveP-TOPk is reported in this paper is the mean of AveP-TOPk calculated over all duplicate bug reports as queries. We primarily use AveP-TOP5.

$$AveP\text{-TOP}_k(Q) = \frac{\sum_{i=1}^{|Q|} in_k(i) \cdot \sum_{j=1}^{i} \frac{in_k(j)}{i}}{\sum_{j=1}^{|Q|} in_k(j)} \tag{3}$$

MMRTOP5: AveP-TOP5 allows for multiple queries to be correct but does not tell us the first query to be correct. This means AveP-TOP5 does not clearly tell us the rank of the successful queries. An alternative to AveP-TOP5 is *MRRTOP5*. MRRTOP5 calculates the mean reciprocal rank of the first TOP5 query to score a *hit*: a duplicate bug report in the TOP5 results. Thus for each bug report the reciprocal rank of the first TOP5 duplicate found is recorded and then the mean is taken for all bug reports. If there is no TOP5 hit, a 0 is returned instead. This score highlights the earliest possible moment that a user paying attention to the top 5 ranked results would find a duplicate bug report.

$$rank^{-1}_{TOP5}(Q) = \begin{cases} \forall q \in Q \neg TOP_5(q) & 0 \\ otherwise & \frac{1}{\min(\arg\max_{j=1}^{|Q|} TOP_5(q_j))} \end{cases}$$

$$MRRTOP5(R) = \frac{1}{|R|} \sum_{i=1}^{|R|} rank^{-1}_{TOP5}(Q_i)$$

Where R is all bug reports, and Q_i is the set of queries of bug report r_i and $r_i \in R$. In cases where the MRRTOP5 is not 0 one can take the reciprocal $MRRTOP5^{-1}$ to find the mean rank at which the duplicate could be found where there is a duplicate to be found.

We report MRRTOP5^{-1} as a convenience. MRRTOP5^{-1} is convenient because it tells you the average number of queries or words to get an answer, but due to its unbounded nature it makes it inappropriate for performance analysis. Because MRRTOP5^{-1} is unbounded we report it in tables but do not rely on it for graphing and statistical analysis—it is a convenience.

3.3.3 Interpretation of metrics

All of these measures, except for MRRTOP5^{-1}, range from 0 to 1. A score of 1 means accurate or perfect query results. While 0 typically means the query did not return the appropriate result.

For TOPk results the value is a mean of how many queries had duplicates in the top k results. TOPk results have limited nuance as they do not tell how soon a hit was found, they just threshold when a hit was found. TOPk are included because they are easy to interpret and when a continuously querying implementation is deployed a small number of examples are typically shown anyways ($k = 5$).

For MAP and MRR they can mostly be interpreted reciprocally. If the number of duplicates for a bug report is 1 then MAP and MRR are essentially the same measure. If there are more results then MAP can be slightly larger than MRR. For MAP, MRR, and AveP-TOP5 and MRRTOP5 the closer to 1 the better the result, the closer to 0 the worse the result. Typically one can interpret these results reciprocally. So a 0.33 MAP or MRR typically means the duplicates were in the top 3 results. Where as 0.2 indicates top 5, 0.1 top 10 and 0.05 top 20. Although for MAP and MRR these imply average performance and do not give an indication if the continuously querying implementation returned duplicate matches sooner.

AveP-TOP5 and MRRTOP5 use the TOP5 measure to help determine the rank of the successful queries. AveP-TOP5 acts much like MAP. It accepts the reality that there is more than 1 duplicate bug report for some bug reports as well there will often be more than 1 successful query a from continuously querying implementation. But it too can be interpreted reciprocally. An AveP-TOP5 of 0.2 typically indicates that with the first 5 queries (the first 5 words) a duplicate bug report appeared in the TOP5 results. Where as MRRTOP5, acts like MRR, of 0.2 means that the 5th word was typically the first introduction, on average, of a TOP5 result with a duplicate bug report. An marginally acceptable MRRTOP5 or AveP-TOP5 should be greater than the reciprocal of our word limit ($25^{-1} = 0.04$), above 0.1 would indicate better than the first 10 words.

In the evaluation we provide *Old MAP* that is MAP classically applied across all of the queries where each bug report is a single query and it's average precision is calculated, and then the mean of all those average precisions of bug report queries are taken. Old MAP is meant to represent the baseline MAP score you can expect if you have all of the information of the bug report. Old MAP lets us compare against other techniques but it is irrelevant to actual continuously querying evaluation performance but serves to remind us of classical bug-deduplication performance.

For *simplicity*, average TOP1 and TOP5 makes a lot of sense to evaluate queries with as they are realistic about users ability to read results. For realism AveP-TOP5 and MRRTOP5 make more sense as they give more weight to continuously querying implementations and algorithms who return good TOP5 results sooner more than those who take longer to produce TOP5 results.

One aspect we did not measure but is related to effort is that we do not have empirical measures of user performance evaluating bug reports to determine how many bug reports need to be evaluated. That is site specific and it depends on what is shown to the user and how such a system is configured. We suggest that if you show 5 bug report titles to the user, the user will read, for bug reports that have a duplicate, $b_1 \cdot 5 \cdot \text{MRR-TOP5}^{-1}$ bug report titles, and if bug report does not have a duplicate they will read up to $b_1 \cdot 5 \cdot 25$ distinct bug reports—assuming 25 words is the limit. We suggest b_1 as a coefficient to represent duplication of results or willingness of a participant to read a bug-report.

4 Experiment methodology

Thus we want to demonstrate the performance of naive continuously querying implementations to show if continuously querying works, that continuously querying can stop duplicate bug reports before they start. By naive we mean a simplified and easy to replicate continuously querying implementation using basic IR tools such as TF-IDF and cosine distance.

Thus the research questions that the following experiments seek to answer are:

- RQ1: How well does our continuously querying implementation perform?
- RQ2: What percentage of duplicate bug reports could be stopped before they start?
- RQ3: Does stemming of tokens matter with respect to continuously querying?
- RQ4: How important is the bug report title to continuously querying?

The validation context and method we use is the third context and method we described in Section 3.2: historical split validation. Historical split validation is used to be fair to future continuously querying implementations, which might use models that are slow to train (e.g., LDA-based (Alipour et al. 2013; Klein et al. 2014; Nguyen et al. 2012) continuously querying implementation models might take some time to retrain), to reduce evaluation time, and to ensure we aren't producing an evaluation based on time-traveling. We will use 100 splits, inducing 100 train and evaluation loops. 100 is chosen because it gives enough granularity and enables any 1% split evaluation. Thus per each split, all bug reports made before the split are trained upon and all duplicate bug reports after the split are evaluated upon if their duplicate exists in the training set. This ensures that we do not learn from the future, we train solely on the past and test solely on the future, 100 times. We rely on the annotations within the bug report repository to determine if bug reports are duplicates of each other. This means we trust the developers and that we only have true-duplicate bug reports marked, and not any true-not-duplicate bug reports.

In the prior Section 3.3 we argued for 25 words as the maximum threshold for number of words continuously queried. We choose this value because it is a few times the average bug report title yet smaller than an entire bug report. Perhaps the user will give up on continuously querying by that time anyways. Regardless the evaluation will be, a bug report from the test set that is a duplicate of a bug report in the training set will be chosen, and 25 queries will be made from it's first 25 words in increasing length ranging from 1 word to 10 words to 25 words. Each of these sequences of words will be used to query the IR model to retrieve the duplicate bug reports. The results of each of these queries will be recorded and TOPk evaluation, MAP, MRR, and AveP-TOPk evaluations will be applied.

The actual words we use as queries is the sequence of words formed by concatenating the bug report title with the bug report description. The documents queried *could* be the concatenation of bug report titles, bug report description, and bug report comments. *In this paper* we concatenate titles and description, not the comments, but depending on the model, the training of a model can be executed differently. Future models using multiple fields for continuously querying should be considered, like BM25F models (Sun et al. 2010, 2011).

Thus we make the assumption that the order that one types in a bug report is title first, followed by the body. We also assume the the final text of the bug report is the text that would be used in continuously querying, but the text written with continuously querying available to the end-user might be different than the text of a full bug report in practice.

4.1 Experimental naive continuously querying implementation

In this paper we provide a naive version of continuously querying. We expect there are more intelligent ways to implement continuously querying that perform better in terms of run-time and information retrieval performance. Our naive continuously querying implementation uses an information retrieval model consisting of TF-IDF transformations of documents, with or without Porter Stemming (Porter 1980), with stop word removal (Manning and Schütze 1999), with digits stripped, with lower casing of alphabetical characters, and no pruning of infrequent and frequent vocabulary. Finally we use the cosine distance between TF-IDF vectors as our method of ranking similar documents to a query document. *In short, we use a TF-IDF and cosine distance indexing implementation from Gensim* (Řehůřek and Sojka 2010). Gensim is a python-based natural language processing and topic modelling library. The TF-IDF we use weights each term in a document as $tf_i \cdot log_2 \frac{|D|}{df_i}$ and then normalizes the document by unit length (Řehůřek and Sojka 2018). This would be similar to the $VSM_{natural}$ model used by Wang et al. (2014). Figure 5 depicts how a query is transformed by this IR model for querying.

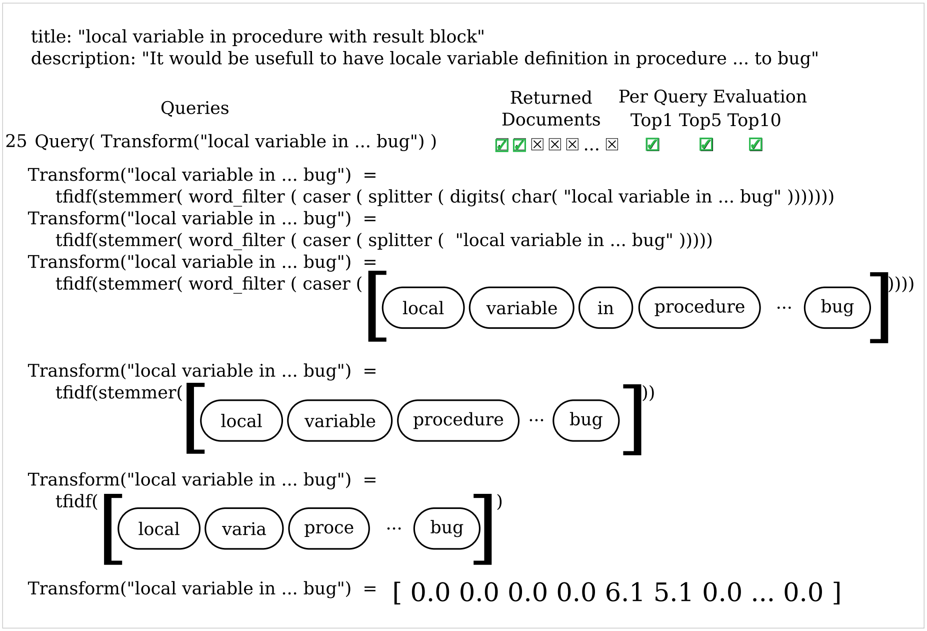

title: "local variable in procedure with result block"
description: "It would be usefull to have locale variable definition in procedure ... to bug"
Queries
Returned Documents
Per Query Evaluation
Top1 Top5 Top10
25 Query(Transform("local variable in ... bug"))
Transform("local variable in ... bug") = tfidf(stemmer(word_filter (caser (splitter (digits(char("local variable in ... bug")))))))
Transform("local variable in ... bug") = tfidf(stemmer(word_filter (caser (splitter ("local variable in ... bug")))))
Transform("local variable in ... bug") = tfidf(stemmer(word_filter (caser ([local variable in procedure ··· bug]))))
Transform("local variable in ... bug") = tfidf(stemmer([local variable procedure ··· bug]))
Transform("local variable in ... bug") = tfidf([local varia proce ··· bug])
Transform("local variable in ... bug") = [0.0 0.0 0.0 0.0 6.1 5.1 0.0 ... 0.0]

Fig. 5 The NLP transformation chain of a text query (from Fig. 3) to a TF-IDF query

We do not reject or threshold query results that are not similar enough according to a similarity score, we rely solely on rank. If one wanted to reduce false positives one could rely on the similar score (normalizing appropriately for query size).

4.2 Datasets

In this paper we use 12 different issue/bug tracker repositories. These datasets are described in Table 1. The dataset used in this paper is available online.[2] The code to run the experiments on this data is available online as well.[3] Table 1 shows the projects, their size in bug reports, duplicate bug reports, and clusters of duplicate bug reports. Table 1 also shows the mean length in words of the title and the description of the bug reports, and date-range of collection.

The software systems that were mined range from cloud management systems, to mobile operating systems, to office suites. The projects were chosen because prior work used them (Android, Eclipse, Mozilla, and OpenOffice (Aggarwal et al. 2015; Alipour et al. 2013; Sun et al. 2011; Sun et al. 2010) and due to availability of tools to mine their repositories. We also explicitly selected repositories with more than 50 duplicate bug pairs that were not used in prior duplicate bug report detection works. Android is an operating system popular on mobile devices and used to be hosted on GoogleCode. App Inventor For Android is a visual programming language that generates Apps for Android. Bazaar is a version control system much like Git. Cyanogenmod is a fork of the Android operating system without Google proprietary apps. Eclipse is an IDE popular with Java programmers. K9mail is an Android

[2]https://archive.org/details/2016-04-09ContinuousQueryData
[3]https://bitbucket.org/abram/continuous-query

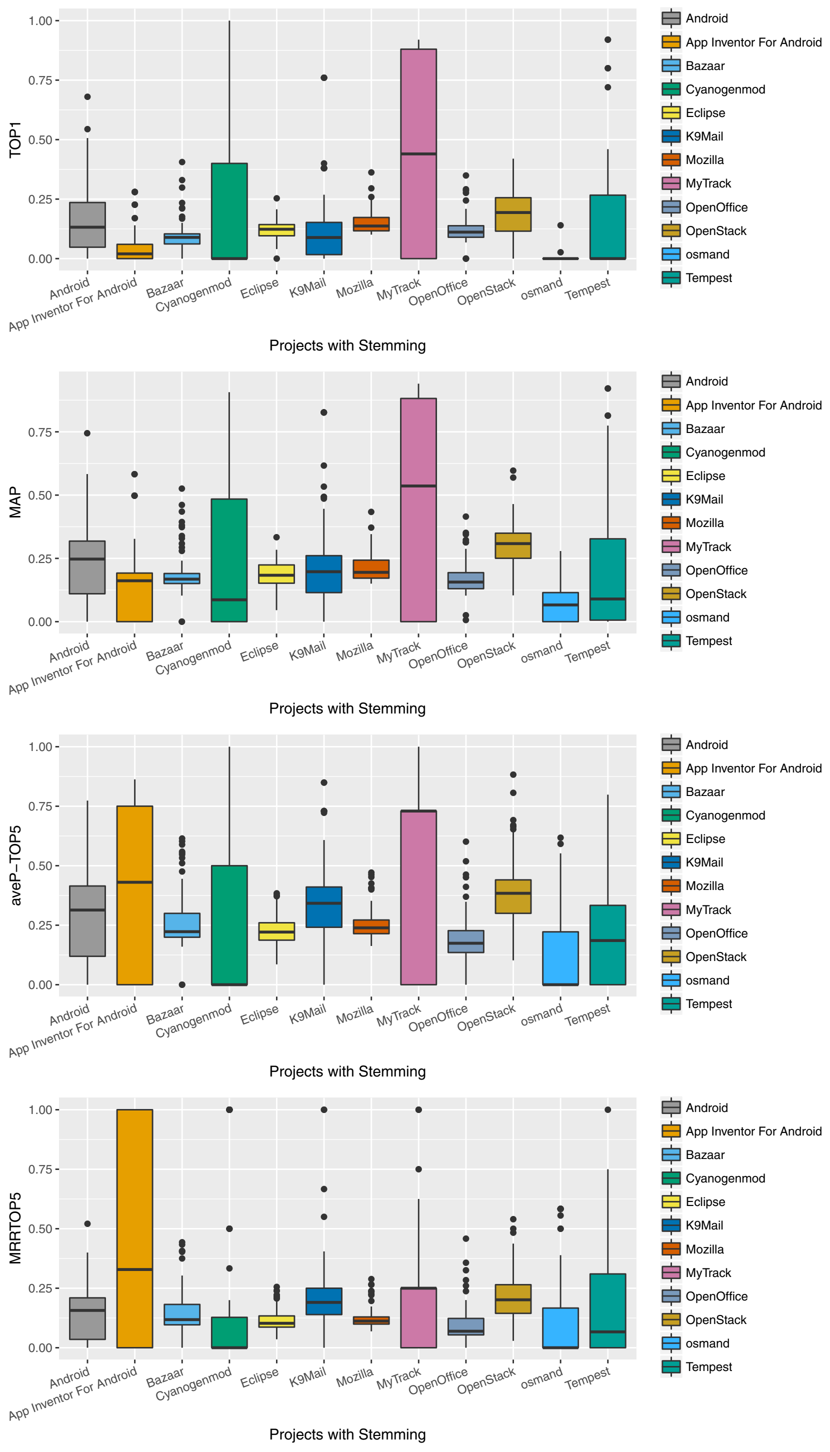


Fig. 6 Boxplots of the distributions of evaluation measures for all 12 projects

app that manages email. Mozilla refers to Mozilla Firefox, a web-browser. MyTrack is a GPS tracking app for Android. OpenOffice is an office suite much like Microsoft Office. OpenStack is a kind of Linux distribution for the cloud to enable management of cloud computers. Osmand is an open-street-maps enabled GPS/mapping program for Android. Tempest is an Integration Test Suite for Openstack. Github issues were not mined due to their lack of consistent markup for duplicate bug reports.

Thus in the next section we will describe the results of our experiments with not-stemming and stemming using a TF-IDF cosine distance model to implement continuously querying.

Table 2 Performance per Project with Continuously Querying

		Old	Continuously Querying							
Project	Exp	MAP	MAP	MRR	TOP1	TOP5	TOP10	aveP-TOP5	MRR-TOP5	MRR-TOP5^{-1}
Android	Stemming	0.240	0.223	0.231	0.150	0.300	0.404	0.274	0.141	7.099
Android	NoStem	0.218	0.194	0.204	0.116	0.294	0.393	0.290	0.164	6.093
App Inventor	Stemming	0.205	0.143	0.197	0.056	0.426	0.532	0.408	0.462	2.165
App Inventor	NoStem	0.140	0.146	0.201	0.046	0.455	0.536	0.441	0.465	2.149
Bazaar	Stemming	0.251	0.185	0.185	0.094	0.278	0.376	0.265	0.148	6.749
Bazaar	NoStem	0.253	0.195	0.195	0.109	0.281	0.360	0.276	0.156	6.424
Cyanogenmod	Stemming	0.286	0.239	0.247	0.205	0.264	0.352	0.249	0.158	6.310
Cyanogenmod	NoStem	0.262	0.179	0.188	0.139	0.260	0.284	0.246	0.172	5.800
Eclipse	Stemming	0.306	0.187	0.190	0.124	0.255	0.316	0.231	0.117	8.517
Eclipse	NoStem	0.301	0.184	0.185	0.119	0.248	0.314	0.231	0.130	7.672
K9Mail	Stemming	0.295	0.226	0.230	0.130	0.348	0.439	0.337	0.207	4.835
K9Mail	NoStem	0.265	0.232	0.237	0.123	0.373	0.441	0.348	0.190	5.261
Mozilla	Stemming	0.288	0.216	0.220	0.152	0.288	0.359	0.258	0.124	8.086
Mozilla	NoStem	0.270	0.199	0.203	0.138	0.263	0.334	0.236	0.116	8.657
MyTrack	Stemming	0.158	0.495	0.495	0.439	0.563	0.582	0.496	0.219	4.562
MyTrack	NoStem	0.165	0.455	0.458	0.398	0.515	0.529	0.453	0.207	4.829
OpenOffice	Stemming	0.248	0.170	0.175	0.121	0.220	0.275	0.196	0.098	10.211
OpenOffice	NoStem	0.270	0.176	0.182	0.130	0.224	0.273	0.196	0.095	10.565
OpenStack	Stemming	0.435	0.302	0.302	0.184	0.436	0.484	0.396	0.214	4.682
OpenStack	NoStem	0.444	0.311	0.311	0.202	0.441	0.500	0.382	0.221	4.518
osmand	Stemming	0.091	0.078	0.078	0.002	0.093	0.280	0.114	0.101	9.866
osmand	NoStem	0.118	0.103	0.103	0.005	0.208	0.474	0.232	0.180	5.554
Tempest	Stemming	0.272	0.184	0.184	0.144	0.206	0.265	0.198	0.156	6.411
Tempest	NoStem	0.255	0.177	0.177	0.133	0.197	0.274	0.219	0.175	5.716
Average	Stemming	0.256	0.221	0.228	0.150	0.306	0.389	0.285	0.179	6.624
Average	NoStem	0.247	0.213	0.220	0.138	0.313	0.393	0.296	0.189	6.103
Average	Both	0.252	0.217	0.224	0.144	0.310	0.391	0.291	0.184	6.364

Old MAP is not Continuously Querying. Gensim's default classic TDF-IF Cosine Distance implementation was used

5 Experiment results

Using the maximum query size of 25 words, and treating for Porter stemming and no stemming we generated 2400 TF-IDF cosine distance models from the 100 splits for each of the 12 evaluated projects. As there are numerous test sets, training sets, and models to build and evaluate, we decided to limit our experiment's breadth by focusing on stemming versus not-stemming because it potentially has the most impact. We could also experiment with other parameters such as digit stripping, lower casing, and pruning of vocabulary, but stemming seemed the most worthwhile to explore due to the technical language used in software engineering whereby stemming could mangle meaningful terms. Stemming could help to reduce vocabulary greatly or it could hide and mangle domain specific terms, reducing query result quality. We evaluated continuously querying queries of duplicate bug reports on these models and aggregated the results for discussion in this section. Figure 6 depicts the performance of Continuously Querying over different projects.

5.1 RQ1: How well does our continuously querying perform?

We sought to evaluate an continuously querying implementation's performance. This implementation relied on Gensim's TF-IDF implementation for document indexing and querying. Using 100 splits per project, we queried for duplicate bug reports using continuously querying queries built from the first 25 words of a bug report, and tallied up the results in Table 2. The performance of this continuously querying implementation is somewhat sensitive to the project under test. The first MAP column of Table 2, "Old MAP", is the MAP score calculated when the only query is the entire bug report document, rather than the 25 continuously querying queries. "Old MAP" serves as a reference to how much information is not in the first 25 words of a bug report. In most cases "Old MAP" is greater than MAP. For all evaluation measures across stemming and not-stemming, the paired Wilcoxon signed rank test shows that the old-style entire-document single query results are statistically significantly different ($\alpha = 0.05$) than the continuously querying results. Thus this continuously querying implementation does not have as good performance as 1 single query, but it still has enough performance that it can be used to prevent duplicate bug reports. Figure 7 summarizes graphically the difference between entire document querying and the continuously

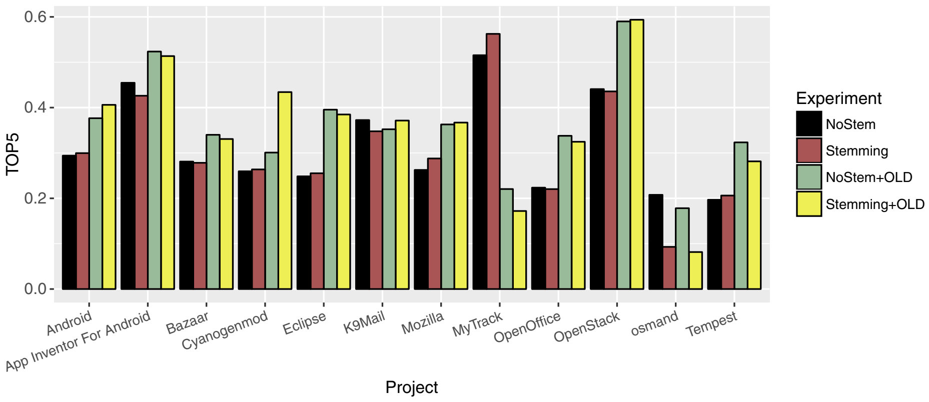


Fig. 7 TOP5 score calculated from the entire document, compared against TOP5 score calculated from TF-IDF gensim continuously querying implementation

querying queries up to the first 25 words of the document. We can see that query performance improves with the full document, but it required the user to write the entire document. This is to be expected as larger document gives more chance for relevant terms.

Based on Table 2 we can see that different projects such as OpenOffice and Android have very different responses. Also the evaluation measures tend to decrease from higher values with the lower ordered splits and lower values with the later splits. This is likely

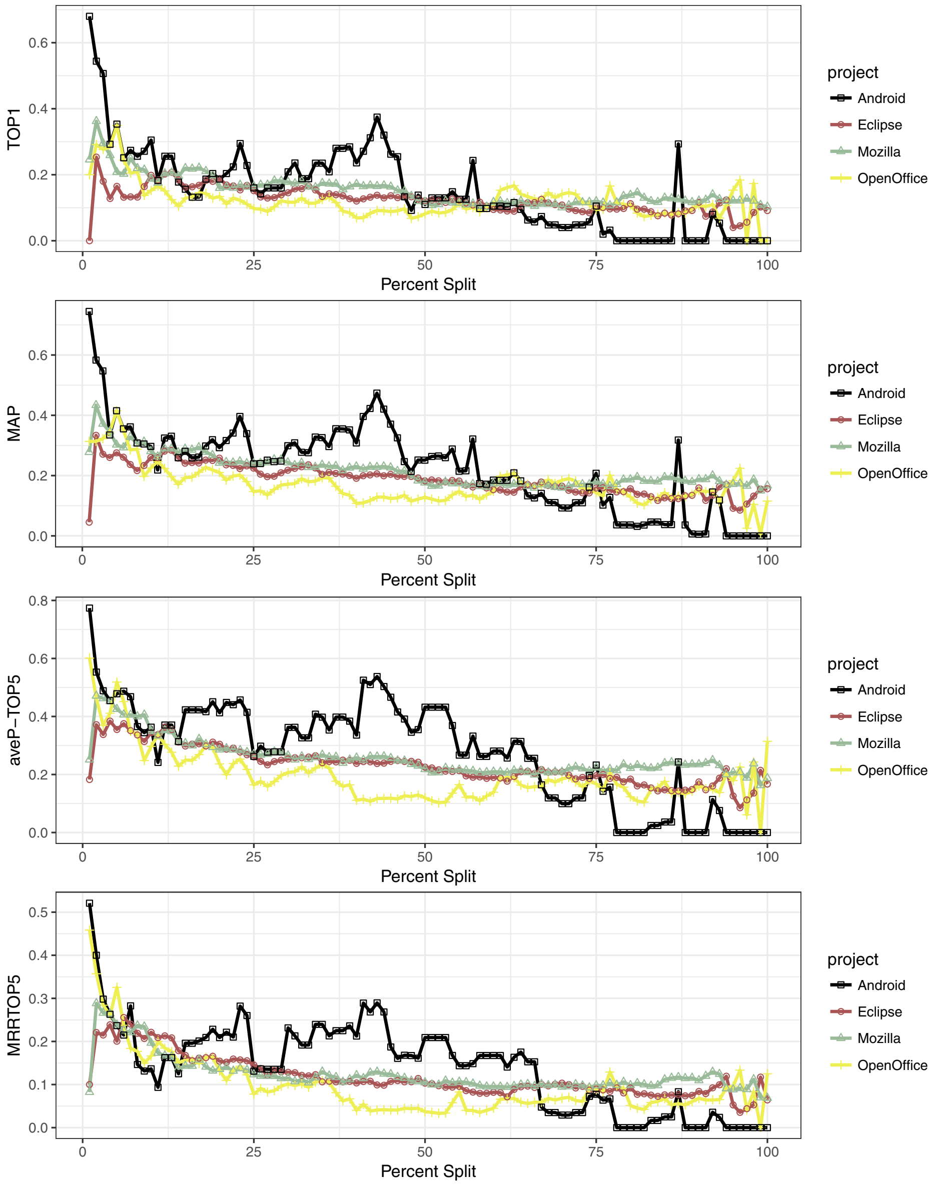


Fig. 8 Evaluation metrics, TOP1, MAP, aveP-TOP5, MRRTOP5 performance over 100 splits across 4 different projects

a consequence of available pool of duplicate bug report queries, but also left biasing of the data because duplicate bug reports do not exist in the future. In Fig. 8 we can see that Android exhibits this right tail phenomenon where there are less bug reports to be duplicates with, as the duplicates are probably coming in the future: it takes time to make duplicate bug reports.

The Android results in Fig. 8 are interesting because if we compare AveP-TOP5 or TOP1 to MAP we can see that MAP results after the 75% split are below 0.1, the rank of 10. Those MAP results are essentially unrealistic query results, with the right answer ranked lower than rank 10 where users might not find them.

We ask, is the full document necessary? What about 1 query with only 25 words, rather than 25 queries or the full document? We assumed that the continuously querying implementation would do worse than 1 query of 25 words because the continuously querying method asks many queries that are lack enough information to resolve. Furthermore we assumed that "Old MAP" would be greater than MAP at 25 words as it had access to more information. 25 word MAP could be greater than "Old MAP" when the project had good quality titles. We investigated and compared "Old MAP" to a single query of the first 25 words. The behaviour observed was relatively consistent: "Old MAP" is usually larger than the MAP of 25-word queries. 25-word queries typically have a better MAP than the continuously querying implementation, much like "Old MAP". There are some differences, MyTrack 25-word queries work better than "Old MAP" and are slightly better in performance than our continuously querying implementation. The Wilcoxon signed rank test (a paired test) suggests that 25 words is different from entire document in terms of MAP ($p = 0.0164$ with $\alpha = 0.05$) but the 95% confidence intervals $[-0.0332, 0.0487]$ do straddle 0. If we exclude MyTrack then the p-value of the Wilcoxon signed rank test is 0.00177 (significant) while the 95% confidence interval is purely negative, $[-0.0415, -0.0211]$, suggesting that full document performance is superior to 25 words in most cases. A single 25 word query achieves better MAP scores than 25 continuously querying queries (Wilcoxon signed rank p-value of $2.015e-05$ and 95% confidence interval of $[0.0280, 0.0512]$).

5.1.1 Causes of poor performance

Some projects did not perform well at all. This has been observed for tuning IR parameters and configurations as well: one size does not fit all Panichella et al. (2016). Osmand does not perform well for TOP1 which is evident in Fig. 4 as it has TOP1 flat performance. We investigated reasons why. First and foremost osmand had a small number of bug reports and duplicates, this will cause some bias as it reduces the number of times we will have a duplicate bug report to query from. Investigating the language of Osmand's bug report, we found many are from mobile device users and the vocabulary in the bug reports can be quite colloquial and less technical. For instance a pair of marked duplicates have titles "checks unckeks in download list" and "Almost impossible to download indexes and voice". The only shared term in the title is "download". Osmand also suffered the most when the bug report description was not used, and only titles were used (discussed in Section 5.4), thus the titles are not very informative. This suggest that disambiguation techniques such as LDA term collocation (Klein et al. 2014) or contextual approaches (Alipour et al. 2013; Aggarwal et al. 2017) could help. Another problem is that for Osmand very few bug reports dominate the top ranked result: 60% of all queries have less than 4 different bug reports appear in the first ranked position. Comparatively as for Bazaar and Tempest, only 43.8% and 44.1% of their queries have less than 4 different bug reports appear in the first ranked position. This domination appears to occur when a term is newly introduced in the model and thus has very high IDF and the same term shows up in the query text—regardless of its relevance. Perhaps discounting or smoothing the IDF of rare terms or removing terms that do not appear in a certain number of documents could improve query performance as to not exaggerate the importance of a new word in the vocabulary of the training set.

5.1.2 RQ1 Summary

What we learn from RQ1 is that typically the performance of an continuously querying implementation in the first 25 words is worse than querying with the entire document, but that up to the first 25 words is sufficient in many cases. This is most likely due to the strength of the keywords used to form the subject or title of bug reports. This is backed up by the MRRTOP5^{-1} measure which shows that a TOP5 result can appear by the 6th word of the query, (see the last column and last rows of Table 2) which is within the range of the title text. Developers and bug reporters should take heed and consider the importance of writing a good bug report title or subject.

5.2 RQ2: What percentage of duplicate bug reports could be stopped before they start?

Based on Table 2 we can see that TOP1 across projects is 0.138 to 0.150. For a large number of queries the first few results will have the duplicate bug ranked first. This means that by 25 words typed in a bug tracker user could notice an immediate duplicate bug. When we move to TOP5 results the value is between 0.306 and 0.313. Indicating about a third of the queries will have TOP5 results within them.

Assuming that once we find a duplicate bug report all of our later bug report queries will have the same response, our AveP-TOP5 indicates that more than 3 words bug report. This is confirmed by the MRRTOP5 results between 0.179 and 0.189 whose inverse, MRRTOP5^{-1} suggests that 6.1 to 6.6 words are needed before a TOP5 hit appears.

But this is actually a skewed result. In projects like OpenOffice approximately 67% of the duplicate bug queries do not find the duplicate in any of the continuously querying queries in the TOP5 results. Per project these failed continuously querying queries make up 16% to 67%. That is 33% to 84% of continuously querying queries (average 42%) in total across all projects depending on the project, will produce duplicate bug reports in their TOP5 query results. MRRTOP5^{-1} shows that for TF-IDF with cosine distance we can achieve on average the first reported duplicate bug report in the TOP5 within 5 to 7 words. This is of course when querying for true duplicate bug reports, globally for all bug reports it depends on the proportion of duplicate to non-duplicate bug reports.

Thus from our data 42 to 43% of duplicate bugs could be expected for projects that initially deploy a continuously querying implementation assuming that the bug reporters can notice the duplicate bug reports via continuously querying. Once continuously querying is adopted these statistics would change due to changing behaviours of users and developers.

5.3 RQ3: Does stemming of tokens matter with respect to continuously querying?

Stemming reduces the size of the vocabulary at a cost of increasing ambiguity. It is argued that stemming is beneficial because it reduces the number of features needed to represent documents. Figure 9 shows stemming across multiple evaluation measures. Many times the stemming performance is greater than the non-stemming performance, but not consistently across all projects and evaluation measures.

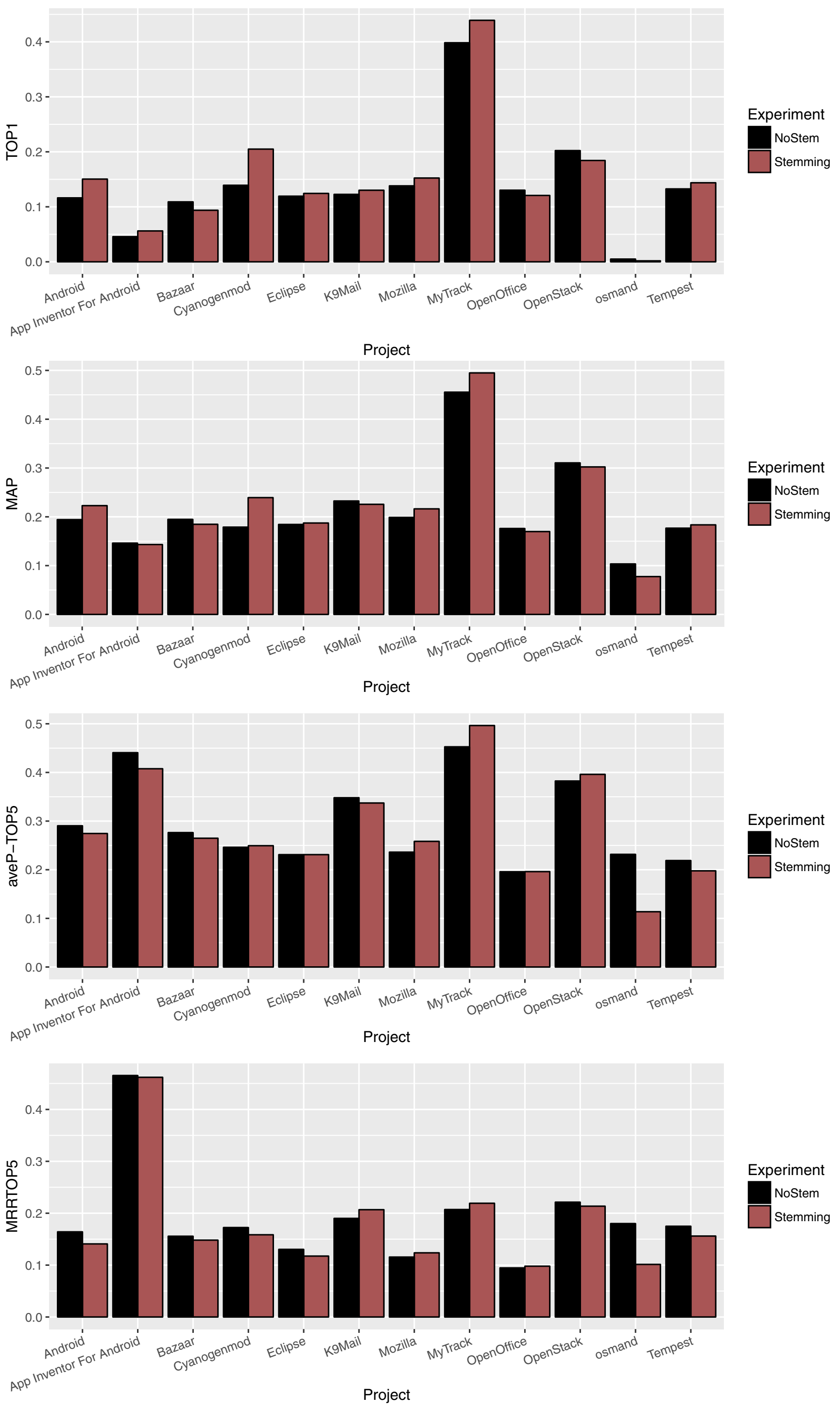


Fig. 9 Average performance of stemming versus not stemming across projects

¿ If we compare the aveP-TOP5 results of all the projects with a Wilcoxon signed rank test we see there's not evidence to show statistical significant difference between stemming and not-stemming performance (p-value of 0.622), with a 95% confidence interval of $[-0.0300, 0.00571]$ for the difference between stemming and not stemming, which shows a slight bias towards not stemming.

Thus globally there's no clear difference, but what about per project? When we drill down and repeat our analysis per each individual project While the plots make it seem like stemming matters, when we compare the 100 aveP-TOP5 values from each of the stemmed and not-stemmed runs per project we find very few results that are not are statistically significant when $\alpha = 0.05$. when using the Wilcoxon signed rank test. For aveP-TOP5, only K9-Mail, MyTracks, Eclipse, and Cyanogenmod did not have evidence to determine if stemming was different than not stemming for a Wilcoxon signed rank test. Most projects showed a difference in a performance, but it is inconsistent. According to their 95% confidence intervals 5 projects (Android, App Inventor, Bazaar, Osmand, and Tempest) do better without stemming, 3 projects (OpenStack, Mozilla, MyTracks) do better with stemming while the remaining projects (Eclipse, K9-Mail, OpenOffice, Cyanogenmod) have confidence intervals that straddle 0. Note that a signed test such as the Wilcoxon signed rank test has better statistical power than it's unpaired cousin the Wilcoxon rank sum test so results can be significant even if the confidence interval crosses 0.

We investigated osmand and MyTracks, 2 projects with the most extreme response to stemming in terms of AveP-TOP5 performance. One example from MyTracks is a pair of duplicate bug reports which mention "rotating" and "rotation". When stemmed these bug report match well, when not stemmed it takes more almost 17 more words to find a duplicate report in the TOP5 results. For osmand stemming causes confusion between features such as routing and routes, which sound similar but are very different semantically. Both will be stemmed to "rout". Stemming doubled the document frequency of "rout" thus reducing its importance during similarity. This is disastrous when the second word of the query is the operational word: the duplicate bug reports had the titles of "Offline Routing: take care of nodes with barrier=xxx" and "Offline Routing: ignoring access restrictions". At split 62 of 100, the difference between the unstemmed and stemmed query can be a TOP5 hit at two words unstemmed and 0 TOP5 hits by 25 words stemmed.

Thus stemming can help resolve similar concepts, but if concepts are named similarly yet are distinct stemming can confuse the two concepts. If two concepts are combined then their document frequencies increase and that term will be of less importance.

Thus we conclude that stemming typically has an effect on a project, but across projects it is not a clear effect and it is not consistently statistically significant. This echos the work of Panichella et al. (2016) who showed that optimal IR configurations differ across projects.

5.4 RQ4: How important is the bug report title to continuously querying?

We wanted to investigate how meaningful the title was to our continuously querying implementation. Thus we compare training and querying solely on titles versus training and querying on both title and description. Rerunning the experiment and training solely on titles we found that for most projects the performances in terms of MRRTOP5 and AveP-TOP5 decreased.

This decrease was insignificant according to a Wilcoxon signed rank test with p-value of 0.509, greater than our $\alpha = 0.05$. The 95% confidence interval of the difference in AveP-TOP5 and MRRTOP5^{-1} between title and description and just title was [-0.00560,0.0262] and [-0.300, 1.88]. Effectively this means that just the titles can work for continuously querying, but at a cost of -0.3 to 1.88 words worth of the queries, but it depends on the project.

Few projects had a large change in MRRTOP5^{-1} performance. Projects that had improved consistent performance using titles, across stemming and not stemming included: Tempest, OpenStack, OpenOffice, MyTrack and Eclipse. Projects that did have consistent drops in performance using titles across stemming and not stemming included: AppInventor, Bazaar, Cyanogenmod, and Osmand. The lack of a clear significant signal suggests that performance for using only titles depends on the project itself. Otherwise the gain for using titles and description over just titles was about 1 queries (MRRTOP5^{-1} mean difference 0.698).

As there was some difference in stemming we checked if aveP-TOP5 was affected by stemming by comparing the distributions with the Wilcoxon signed rank test, which concluded that for title-only queries there was no evidence to support that stemming or not stemming were significantly different (p-value of 0.6221) with a 95% confidence interval of [-0.0300, 0.0564]. We conclude that for title-only queries stemming has little effect.

Cyanogenmod and Osmand had particularly poor performance using only titles: each respectively had a MRRTOP5^{-1} difference of 2.97 and 11.09 with no stemming. For osmand this suggests that osmand bug report subjects and titles are not descriptive enough to find similar duplicates and thus the description needs to be relied upon.

We conclude that title-only querying does perform relatively well and if run-time performance was an issue and the scale was small enough title-only querying can be very efficient and potentially done client side. Although the gain in run-time performance comes at a penalty in terms of general retrieval performance and the number of words till a duplicate is returned.

6 BM25 Evaluation

How does this baseline technique stand up to the techniques used in other research? Sun et al. (2010, 2011) proposed REP which used multiple fields, bi-grams of titles, and text from the description combined with BM25F a multiple field form of BM25 and Okapi BM25 (Robertson et al. 1995). BM25 evaluates queries much like TF-IDF and cosine distance. It has 2 parameters k and b_1. Since our continuously querying implementation did not have multiple fields yet to rely upon we could not use the more complicated parts of REP. Thus we made a REP-inspired continuously querying implementation using BM25 instead of TF-IDF and cosine distance.

We repeated the same methodology as the TF-IDF gensim continuously querying and executed continuously querying queries across all of the projects in the dataset. We used Gensim's implementation and we used unoptimized defaults of $k_1 = 0.7$ and $b = 1.25$ from Gensim.

6.1 BM25 performance

Unfortunately BM25's performance was lackluster compared to our TF-IDF implementation. Table 3 shows that the average query taking 10 to 12 words to find a duplicate bug report. The AveP-TOP5 performance of 0.123 to 0.141 was worse than the 0.285 to 0.296 performance of the initial TF-IDF implementation. But even in the case of Old Map the BM25 suffered heavily with Old MAP values of to 0.112 to 0.114. A Wilcoxon signed rank

Table 3 Performance per project with BM25 gensim continuously querying implementation. Old MAP is not continuously querying

		Old	Continuously querying							
Project	Exp	MAP	MAP	MRR	TOP1	TOP5	TOP10	aveP-TOP5	MRR-TOP5	MRR-TOP5^{-1}
Android	Stemming	0.113	0.035	0.041	0.022	0.051	0.074	0.087	0.075	13.314
Android	NoStem	0.079	0.037	0.045	0.025	0.060	0.075	0.102	0.078	12.828
App inventor	Stemming	0.086	0.104	0.143	0.064	0.200	0.271	0.216	0.098	10.193
App inventor	NoStem	0.029	0.104	0.134	0.078	0.206	0.254	0.228	0.105	9.510
Bazaar	Stemming	0.140	0.086	0.086	0.060	0.110	0.133	0.106	0.053	18.773
Bazaar	NoStem	0.171	0.085	0.085	0.055	0.113	0.142	0.108	0.055	18.085
Cyanogenmod	Stemming	0.058	0.034	0.042	0.036	0.040	0.052	0.040	0.040	25.000
Cyanogenmod	NoStem	0.074	0.045	0.053	0.036	0.079	0.119	0.065	0.048	20.638
Eclipse	Stemming	0.197	0.104	0.105	0.074	0.136	0.169	0.126	0.068	14.643
Eclipse	NoStem	0.202	0.120	0.123	0.090	0.154	0.188	0.148	0.091	11.000
K9Mail	Stemming	0.079	0.038	0.039	0.009	0.052	0.094	0.080	0.059	17.025
K9Mail	NoStem	0.057	0.089	0.091	0.031	0.148	0.186	0.146	0.091	11.009
Mozilla	Stemming	0.113	0.062	0.065	0.043	0.085	0.104	0.087	0.047	21.419
Mozilla	NoStem	0.128	0.078	0.080	0.056	0.101	0.119	0.099	0.052	19.296
MyTrack	Stemming	0.052	0.151	0.153	0.080	0.205	0.306	0.257	0.164	6.103
MyTrack	NoStem	0.069	0.142	0.144	0.074	0.230	0.287	0.279	0.166	6.038
OpenOffice	Stemming	0.089	0.084	0.090	0.072	0.104	0.117	0.099	0.056	17.744
OpenOffice	NoStem	0.107	0.086	0.092	0.071	0.108	0.124	0.106	0.060	16.575
OpenStack	Stemming	0.258	0.128	0.128	0.071	0.201	0.220	0.162	0.082	12.180
OpenStack	NoStem	0.232	0.160	0.160	0.108	0.219	0.269	0.195	0.118	8.455
osmand	Stemming	0.059	0.071	0.071	0.053	0.078	0.079	0.087	0.070	14.227
osmand	NoStem	0.057	0.066	0.066	0.047	0.073	0.082	0.081	0.067	14.940
Tempest	Stemming	0.128	0.066	0.066	0.038	0.085	0.116	0.125	0.095	10.548
Tempest	NoStem	0.140	0.072	0.072	0.039	0.096	0.119	0.137	0.101	9.858
Average	Stemming	0.114	0.080	0.086	0.052	0.112	0.145	0.123	0.076	15.097
Average	NoStem	0.112	0.090	0.095	0.059	0.132	0.164	0.141	0.086	13.186
Average	Both	0.113	0.085	0.091	0.056	0.122	0.154	0.132	0.081	14.142

test shows that BM25 stemmed and not-stemmed AveP-TOP5s are significantly different than TF-IDF with cosine distance ($p = 1.192e-07$ with $\alpha = 0.05$) with a 95% confidence interval of difference between TF-IDF and BM25 of 0.138 to 0.178.

This indicated that perhaps tuning needed to be done. We explored various values for k_1 and saw little difference. Following a grid search of b from 0.0 to 1.0 on App Inventor we found the highest MRRTOP5 of 0.149 at $b = 0.0$ and lowest of 0.103 at $b = 1.0$. The b parameter normalizes for length and by setting b to 0.0 normalization is disabled, but even at $b = 0.0$ the average performance of BM25 versus TF-IDF cosine distance was for MRRTOP5 was 0.149 versus 0.465. Thus tuning can improve BM25 performance, but BM25 did not fair well potentially due to document length which the b parameter was modifying the weight of. An alternative could be to evaluate a newer BM25 derivative.

Table 4 Performance per Project with DüpeBuster ElasticSearch/Lucene Lucene-Score based Continuously Querying Implementation. Old MAP is not Continuously Querying

		Old	Continuously querying							
Project	Exp	MAP	MAP	MRR	TOP1	TOP5	TOP10	aveP-TOP5	MRR-TOP5	MRR-TOP5^{-1}
Project	Exp	MAP	MAP	MRR	TOP1	TOP5	TOP10	aveP-TOP5	first-TOP5	1/1stTop5
Android	Stemming	0.197	0.243	0.276	0.195	0.385	0.465	0.364	0.202	4.947
Android	NoStem	0.211	0.241	0.268	0.175	0.406	0.478	0.380	0.202	4.944
App inventor	Stemming	0.171	0.147	0.184	0.071	0.330	0.530	0.374	0.463	2.158
App inventor	NoStem	0.115	0.173	0.230	0.137	0.344	0.543	0.390	0.401	2.493
Bazaar	Stemming	0.236	0.227	0.227	0.142	0.322	0.397	0.302	0.159	6.285
Bazaar	NoStem	0.242	0.225	0.225	0.148	0.312	0.380	0.298	0.160	6.232
Cyanogenmod	Stemming	0.209	0.169	0.174	0.107	0.262	0.294	0.233	0.134	7.480
Cyanogenmod	NoStem	0.222	0.222	0.223	0.144	0.309	0.345	0.265	0.138	7.252
Eclipse	Stemming	0.292	0.230	0.233	0.168	0.307	0.365	0.290	0.145	6.899
Eclipse	NoStem	0.304	0.234	0.237	0.170	0.316	0.366	0.299	0.158	6.314
K9Mail	Stemming	0.210	0.200	0.205	0.117	0.302	0.378	0.315	0.186	5.364
K9Mail	NoStem	0.224	0.264	0.269	0.200	0.360	0.414	0.347	0.179	5.574
Mozilla	Stemming	0.271	0.228	0.232	0.164	0.312	0.372	0.293	0.144	6.964
Mozilla	NoStem	0.283	0.232	0.236	0.172	0.310	0.366	0.296	0.145	6.905
MyTrack	Stemming	0.114	0.458	0.459	0.403	0.531	0.586	0.490	0.245	4.082
MyTrack	NoStem	0.127	0.434	0.436	0.394	0.465	0.531	0.428	0.194	5.158
OpenOffice	Stemming	0.189	0.186	0.194	0.146	0.240	0.296	0.226	0.109	9.207
OpenOffice	NoStem	0.209	0.182	0.189	0.142	0.243	0.294	0.232	0.120	8.362
OpenStack	Stemming	0.402	0.316	0.316	0.193	0.481	0.530	0.421	0.239	4.180
OpenStack	NoStem	0.354	0.331	0.331	0.218	0.472	0.531	0.418	0.240	4.158
osmand	Stemming	0.125	0.126	0.126	0.028	0.286	0.384	0.294	0.189	5.284
osmand	NoStem	0.090	0.152	0.152	0.058	0.289	0.397	0.313	0.218	4.592
Tempest	Stemming	0.215	0.145	0.145	0.091	0.189	0.290	0.276	0.198	5.056
Tempest	NoStem	0.208	0.161	0.161	0.099	0.220	0.310	0.301	0.194	5.155
Average	Stemming	0.219	0.223	0.231	0.152	0.329	0.407	0.323	0.201	5.659
Average	NoStem	0.216	0.238	0.246	0.171	0.337	0.413	0.331	0.196	5.595
Average	Both	0.217	0.230	0.239	0.162	0.333	0.410	0.327	0.198	5.627

7 Industrial evaluation

How would a continuously querying implementation perform in actual software development? Can we design systems capable of handling the sheer number of queries that continuously querying of bug reports calls for?

We have deployed continuously querying through our industrial continuously querying implementation called Bug-Party/DüpeBuster[4]. What we have found is that providing a RESTful webservice that can respond to continuously querying queries on bug reports is quite possible with modern NLP indexing. The TF-IDF continuously querying implementation depicted in the paper is not the same as DüpeBuster, which uses elasticsearch's and Lucene's TF-IDF implementation with Lucene and ElasticSearch scoring. We integrated DüpeBuster with existing bug trackers to enable continuously querying. By injecting some JavaScript into a bug tracker's webpage one can add functionality to enable text input fields to provide continuously querying queries to the DüpeBuster continuously querying webservice. The DüpeBuster UI is similar to the UI depicted in Fig. 1. As the developer types in a bug report suggestions pop-up beside their text input area. This is an *online* behaviour. The user's browser queries the webservice and then the suggestions are laid out beside the user's bug text input area. They can mouse over the suggestions to get more information or they may click one of them to open a new tab with that potential duplicate bug report. One difference between the DüpeBuster's UI and the continuously querying implementation described previously is that when DüpeBuster is deployed the Lucene score of similarity is shared and used as a threshold to determine whether or not a duplicate suggestion should be shown to the user. Thereby reducing false positives, but obviously reducing recall. In a deployed system it can make sense to engage in thresholding of results based on score or confidence.

We worked with BioWare to refine and deploy Bug-Party/DüpeBuster and it was deployed within the bug trackers for a few of their games. As testers and developers type a bug report into the internal bug tracking tools, the current report's title and description would be sent to DüpeBuster which would reply with possible suggestions. These suggestions would be displayed underneath the typed text and the bug reporter could click on any of the suggestions to open a new tab containing that bug report.

7.1 DüpeBuster performance

The performance of DüpeBuster is measured in Table 4. It follows the same evaluation methodology as the experiments in Table 2. The main difference is that ElasticSearch has to be told to refresh its index after bug reports are loaded individually in DüpeBuster. The 95% confidence interval of mean difference between the gensim TF-IDF continuously querying implementation and the DüpeBuster continuously querying implementation for AveP-Top5 is 0.0216 to 0.0595, about a 0.0409 difference in performance. A Wilcoxon signed rank test between gensim and DüpeBuster performance indicates a significant difference with a p-value of 0.00158.

The DüpeBuster implementation performance is slightly better on average than the Gensim implementation.

One difference between the Gensim implementation and DüpeBuster implementation was that stemming had limited effect on in performance according to bootstrapped confidence intervals of difference of means between not stemming and stemming at 95% was between 0.00610 and 0.0208. Yet a Wilcoxon signed rank test on the per project means had a p-value of only 0.0923 indicates that the difference is insignificant as it is above the α of 0.05.

If we look to Tables 2 and 4 the mean difference in MRR-TOP5 performance is about 0.7 words. The gensim based prototype returns a successful hit in the TOP5 1 query/word

[4]To install DüpeBuster visit https://bitbucket.org/abram/bugparty-docker and https://bitbucket.org/abram/bugparty/.

later than DüpeBuster. Performance is probably different for a few practical reasons. First of all, DüpeBuster was made earlier than the TF-IDF continuously querying implementation demonstrated here, second it only returns 25 results, so MAP and MRR scores will be cut off at 25 results for all queries. Thirdly, when DüpeBuster is deployed 2 queries are made against the subject and description of the bug report and the results are interleaved. Fourthly, DüpeBuster uses Lucene which might do far more work than Gensim. Since this evaluation engages in concatenation of bug report subject and description we only queried and ranked against the description match. Furthermore ElasticSearch employs Lucene's default language model and thus would not benefit much from pre-stemming as it was expecting English words—although ElasticSearch and Lucene are quite configurable. The benefit of using an ElasticSearch-based backend is that the index is flexible, it updates rapidly, it can be clustered, and it has performance that allow numerous concurrent clients to make multiple queries per second.

7.2 Interviews with developers and testers

At BioWare we sent out interview invites to more than *80* developers and testers for a variety of BioWare games that were using Bug-Party/DüpeBuster. We recruited and interviewed *6* volunteers who used the tool and the bug trackers. While 6 is not enough to describe deployment statistically we will report differences and commonalities that occurred within the interviews. The volunteers ranged from testers to developers, to project managers. Some used the bug tracker all day, some used the bug tracker a few times a week. All of them had encountered and could identify the DüpeBuster.

We asked the interviewees about how bug reports were filed and they responded that testers received formal training and documentation about how to search and how to write bug report titles. A typical process for reporting a bug was to search an bug tracker manually for the bug in question using attributes such as levels, dates, and keywords from the title. If a relevant bug report was not found they would report the bug through BioMetrics where DüpeBuster was installed. As they typed in the bug report DüpeBuster would provide suggestions that could be relevant.

Half (3/6) of the interviewees intentionally and actively used the DüpeBuster to query bug reports. The others claimed it was more passive, they would try to report a bug and if DüpeBuster reported anything they might investigate.

All (6/6) of the interviewees stated that DüpeBuster had prevented a duplicate bug. An interviewee responded, "It's a really good tool and it should be developed. It cuts down on QA time."

Most (5/6) interviewees believed that BioWare saved money due to time saved not writing duplicate bug reports. Of those who said it does not save money tended to have bug reporting workflows where they would query JIRA first and quite thoroughly. Some suggested improvements such as integrating DüpeBuster into tools used earlier in the process.

All (6/6) of the interviewees said it was fast, quick, or fast enough. One interviewee expressed: "It's very fast. Visually, DüpeBuster responds to each word typed within less than a second."

When asked how many words we needed to provide before continuously querying would find an appropriate bug the answers ranged from 2 to 5 words, 3 words, to 4 to 5 words, to the entire title. Many interviewees emphasized the importance of the bug report title.

The interviewees tried to estimate the rate of bug prevention, their responses were: 3 times a month, 2/500, 1/100, 1/50, 1/30, and 30%. So there was a consistent belief that some duplicate bug reports were being prevented. One interviewee was commenting on how duplicates were found by DüpeBuster, “It catches dupes from titles about 50% of the time.” This indicates that the potential claims of bug report prevention presented in this paper could be over-stated yet much of this responding could due to the common process of searching JIRA first by hand then moving on to report the duplicate.

Improvements were suggested by all of the interviewees. They suggested that meta-data such as Bug ID or date, and short summaries were needed as the title was not always enough. Many of the interviewees said not to show closed bugs, or instead to focus on recent bug reports. Some suggested to keep a query dictionary in order to address aliases for terms.

Aliases for terms are important because the name for an asset or entity in a program can be different than the name of the asset or entity in a level or story, so there can be ambiguity in bug reports. The theme of common vocabulary came up numerous times as aliased keywords were perceived to cause problems with bug report search.

Two interviewees were asked if the continuously querying implementation was distracting and they responded that it was easy to ignore when it was no longer relevant. Yet this suggests that continuously querying UI might have to face human limits and different strategies should be employed based on the amount of provided text.

If we relate our 42% duplicate prevention from RQ2 in Section 5.2 with observations from this section we can see that developers perceive that the continuously querying implementation prevents duplicate bug reports, and that it saves money due to preventing duplicates even if that rate of duplicate prevention is low.

Thus we can see that not only is the continuously querying method industrially relevant but it incurs minimal costs for developers to use as it can be integrated into their existing systems. We thank BioWare employees for their participation in this study.

8 Threats to validity

Construct validity faces many threats. First and foremost, do we trust the duplicate bug report data extracted from numerous bug report systems? Do we trust developers to properly mark duplicate bug reports? Even so, at no point do they mark non-duplicate bugs, thus we have no negative examples. Since we only supposedly have true positive examples we can only measure precision and thus are stuck with measures like MAP or MRR.

In our evaluation, We assume that users type in their bug reports in order. We do not know if this is how they compose their bug reports. We assume that users write the title of their bug report first as per our interviews. We also assume that users will read some of the duplicate bug report suggestions yet we cannot present evidence of this yet without more invasive logging. We also did not measure how many bug reports a user would have to read or evaluate to find or not find a duplicate bug reports in a deployed scenario.

Furthermore we assume the words in these bug reports are the words that would be used in continuously querying. An edited bug report might be very different from the text a developer would type if a continuously querying implementation was available to them.

Internal validity is hampered by the lack of negative queries, we only ask positivist queries where we know there is an answer. This means we might have abysmal performance for bug reports that are not duplicates and we might waste the user’s time evaluating non-duplicate bug reports. Internal validity of the industrial evaluation was hampered by the limited number of employees who used the tool and the limited number of volunteers even.

External validity is threatened by the small number of systems and small number of duplicates per system. External validity could be better addressed with more and wider varieties of duplicate bug data. External validity is further hampered by the lack of sampling in terms of the datasets used – they are used opportunistically rather than statistically sampled and this will affect the reliability of the results as well as external validity. Our industrial deployment was only at 1 company.

9 Conclusion

The *Continuously Querying* approach to preventing duplicate bug reports allows users to find duplicate bug reports as they type in their bug report. The string of their in-progress bug report becomes a query to find duplicate bug reports. This is done by continuously querying the bug tracker for duplicate reports with every new word that the user types in, much like search engine suggestions. By rapidly querying the bug tracker, duplicate bug reports may be found and suggested before a user finishes writing a bug report.

Continuously Querying Bug Reports is a new kind of bug deduplication task that has the potential to prevent duplicate before they occur in 42% or more of observed duplicate bug report cases. We demonstrated via a rigorous experiment that continuously querying implementation are effective at finding duplicate bug reports in multiple projects. We created a simple information retrieval model using TF-IDF and cosine distance to find similar documents from our prefixed queries. We found that in general stemming our vocabulary showed inconsistent statistical evidence of improving performance.

Can current state-of-the-art bug report deduplication techniques transition well into continuously querying algorithms and implementations? As this is a new kind of bug deduplication method which needs further research into appropriate IR techniques to retrieve bug reports, we have shared our dataset and source code.[5]

9.1 Future work

As this work introduces continuously querying bug reports, there is only so much can be done in one paper. There are many possibilities for future research on continuously querying and we hope that the availability of our benchmark dataset will attract some new research in this area.

New continuously querying algorithms should be proposed and tested. Yet future work should also consider evaluation in terms of runtime performance, IR performance, and human performance in terms human effort to filter and read these duplicate bug report suggestions. Runtime performance and more importantly human effort deserve more attention than this study gave them.

We employed very naive IR in this paper to implement continuously querying. We used TF-IDF with cosine distance and BM25 in this work, where as other bug deduplication works have used BM25F Sun et al. (2010, 2011). We did evaluate BM25 but have not used BM25F—BM25 on multiple fields.

While some of our results showed that the application of techniques produce conflicting results based on the project, we did not attempt to find or choose near optimal configurations. We were not able to explore the effect of different NLP and IR treatments beyond the use of stemming. Clear follow up studies would be about the effects of different IR and NLP techniques on continuously querying: does stop word removal matter; should entity recognition be employed; is there a benefit to limiting the vocabulary; could character n-grams

[5]Datasets https://archive.org/details/2016-04-09ContinuousQueryData. Code: https://bitbucket.org/abram/continuous-query

perform better; can we improve run-time performance with stateful queries or caching result sets? Continuously Querying research can be bolstered with search based software engineering (Harman et al. 2012), IR search techniques employed by Panichella et al. (2016), and query quality and reformulation techniques proposed by Haiduc (2014).

Furthermore the elephant in the room of continuously querying is that the words used in a bug report are not the same as words used in queries. Thus word filtering, query-word expansion, and query word re-weighting need to be investigated to improve continuously querying implementation performance.

Acknowledgments This work was funded by an NSERC Discovery Grant, NSERC Engage Grant, and a MITACS Accelerate Cluster Grant in conjunction with Bioware ULC. We would also like to thank prior reviewers and Ahmed Hassan.

References

Aggarwal K, Timbers F, Rutgers T, Hindle A, Stroulia E, Greiner R (2017) Detecting duplicate bug reports with software engineering domain knowledge. Journal of Software: Evolution and Process 29:1–15. https://doi.org/10.1002/smr.1821 http://softwareprocess.ca/pubs/aggarwal2017JSEP.pdf E1821 smr.1821

Aggarwal K, Rutgers T, Timbers F, Hindle A, Greiner R, Stroulia E (2015) Detecting duplicate bug reports with software engineering domain knowledge. In: 22nd international conference on software analysis, evolution and reengineering (SANER), 2015 IEEE, pp 211–220. IEEE

Alipour A (2013) A contextual approach towards more accurate duplicate bug report detection. Master's thesis University of Alberta

Alipour A, Hindle A, Stroulia E (2013) A contextual approach towards more accurate duplicate bug report detection. In: Proceedings of the Tenth International Workshop on Mining Software Repositories, pp 183–192. IEEE Press

Arasu A, Babu S, Widom J (2006) The cql continuous query language: semantic foundations and query execution. VLDB J 15(2):121–142

Asaduzzaman M, Roy CK, Schneider KA, Hou D (2014) Cscc: Simple, efficient, context sensitive code completion. In: 2014 IEEE International conference on software maintenance and evolution (ICSME), pp 71–80. IEEE

Bettenburg N, Premraj R, Zimmermann T, Kim S (2008) Duplicate bug reports considered harmful really? In: IEEE international conference on software maintenance, 2008. ICSM 2008, pp 337–345. IEEE

Campbell JC, Santos EA, Hindle A (2016) The unreasonable effectiveness of traditional information retrieval in crash report deduplication. In: International Working Conference on Mining Software Repositories (MSR 2016), pp 269–280. https://doi.org/10.1145/2901739.2901766. http://softwareprocess.ca/pubs/campbell2016MSR-partycrasher.pdf

Chandrasekaran S, Cooper O, Deshpande A, Franklin MJ, Hellerstein JM, Hong W, Krishnamurthy S, Madden SR, Reiss F, Shah MA (2003) Telegraphcq: Continuous dataflow processing. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD '03. ACM, New York, pp 668–668. http://doi.acm.org/10.1145/872757.872857

Chandrasekaran S, Franklin MJ (2002) Streaming queries over streaming data. In: Proceedings of the 28th International Conference on Very Large Data Bases, VLDB '02. VLDB Endowment, pp 203–214. http://dl.acm.org/citation.cfm?id=1287369.1287388

Deshmukh JMAK, Podder S, Sengupta S, Dubash N (2017) Towards accurate duplicate bug retrieval using deep learning techniques. In: 2017 IEEE International conference on software maintenance and evolution (ICSME), pp 115–124. https://doi.org/10.1109/ICSME.2017.69

Google (2016) Google suggestion service https://goo.gl/4sFq8n

Haiduc S (2014) Supporting query formulation for text retrieval applications in software engineering. In: 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014, pp 657–662. IEEE Computer Society. https://doi.org/10.1109/ICSME.2014.117 https://doi.org/10.1109/ICSME.2014.117

Harman M, Mansouri SA, Zhang Y (2012) Search-based software engineering: Trends, techniques and applications. ACM Comput Surv 45(1):11:1–11:61. https://doi.org/10.1145/2379776.2379787. http://doi.acm.org/10.1145/2379776.2379787

Jalbert N, Weimer W (2008) Automated duplicate detection for bug tracking systems. In: IEEE International Conference on dependable systems and networks with FTCS and DCC, 2008. DSN 2008, pp 52–61. IEEE

Kao B, Garcia-Molina H (1994) An overview of real-time database systems. In: Real time computing, pp 261–282. Springer

Klein N, Corley CS, Kraft NA (2014) New features for duplicate bug detection. In: MSR, pp 324–327

Lazar A, Ritchey S, Sharif B (2014) Improving the accuracy of duplicate bug report detection using textual similarity measures. In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp 308–311. ACM

Lukins SK, Kraft NA, Etzkorn LH (2008) Source code retrieval for bug localization using latent dirichlet allocation. In: Proceedings of the 2008 15th Working Conference on Reverse Engineering, WCRE '08. IEEE Computer Society, Washington, pp 155–164. https://doi.org/10.1109/WCRE.2008.33

Manning CD, Schütze H (1999) Foundations of Statistical Natural Language Processing. The MIT Press, Cambridge. http://nlp.stanford.edu/fsnlp/

Nguyen AT, Nguyen TT, Nguyen TN, Lo D, Sun C (2012) 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, pp 70–79. ACM

Panichella A, Dit B, Oliveto R, Penta MD, Poshyvanyk D, Lucia AD (2016) Parameterizing and assembling ir-based solutions for SE tasks using genetic algorithms. In: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016, pp 314–325. IEEE Computer Society. https://doi.org/10.1109/SANER.2016.97

Ponzanelli L, Bacchelli A, Lanza M (2013) Seahawk: Stack overflow in the ide. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13. IEEE Press, Piscataway, pp 1295–1298. http://dl.acm.org/citation.cfm?id=2486788.2486988

Ponzanelli L, Bavota G, Di Penta M, Oliveto R, Lanza M (2014) Mining stackoverflow to turn the ide into a self-confident programming prompter. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014. ACM, New York, pp 102–111. http://doi.acm.org/10.1145/2597073.2597077

Porter M (1980) An algorithm for suffix stripping. Program 14(3):130–137. https://doi.org/10.1108/eb046814 http://www.emeraldinsight.com/doi/abs/10.1108/eb046814

Rakha MS, Bezemer CP, Hassan AE (2017) Revisiting the performance evaluation of automated approaches for the retrieval of duplicate issue reports. IEEE Trans Softw Eng PP(99):1–1. https://doi.org/10.1109/TSE.2017.2755005

Rakha MS, Bezemer CP, Hassan AE (2018) Revisiting the performance of automated approaches for the retrieval of duplicate reports in issue tracking systems that perform just-in-time duplicate retrieval Empirical Software Engineering

Rakha MS, Shang W, Hassan AE (2015) Studying the needed effort for identifying duplicate issues. Empirical Software Engineering pp 1–30. https://doi.org/10.1007/s10664-015-9404-6

Řehůřek R, Sojka P (2010) Software Framework for Topic Modelling with Large Corpora. In: Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks. ELRA, Valletta, pp 45–50. http://is.muni.cz/publication/884893/en

Řehůřek R, Sojka P (2018) models.tfidfmodel — TF-IDF model. https://radimrehurek.com/gensim/models/tfidfmodel.html (retrieved, March 2018)

Robertson S, Walker S, Jones S, Hancock-Beaulieu MM, Gatford M (1995) Okapi at trec–3. In: Overview of the Third Text REtrieval Conference (TREC–3), pp 109–126. Gaithersburg, MD: NIST. https://www.microsoft.com/en-us/research/publication/okapi-at-trec-3/

Rocha H, De Oliveira G, Marques-Neto H, Valente MT (2015) Nextbug: a bugzilla extension for recommending similar bugs. Journal of Software Engineering Research and Development 3(1):1–14

Runeson P, Alexandersson M, Nyholm O (2007) Detection of duplicate defect reports using natural language processing. In: 29th international conference on Software engineering, 2007. ICSE 2007, pp 499–510. IEEE

Sabor KK, Hamou-Lhadj A, Larsson A (2017) Durfex: a feature extraction technique for efficient detection of duplicate bug reports. In: 2017 IEEE international conference on software quality, reliability and security (QRS), pp 240–250. IEEE

Shah MA, Hellerstein JM, Chandrasekaran S, Franklin MJ (2003) Flux: an adaptive partitioning operator for continuous query systems. In: Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405), pp 25–36. https://doi.org/10.1109/ICDE.2003.1260779

Sun C, Lo D, Khoo SC, Jiang J (2011) Towards more accurate retrieval of duplicate bug reports. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, pp 253–262. IEEE Computer Society

Sun C, Lo D, Wang X, Jiang J, Khoo SC (2010) A discriminative model approach for accurate duplicate bug report retrieval. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, pp 45–54. ACM

Sureka A, Jalote P (2010) Detecting duplicate bug report using character n-gram-based features. In: Software engineering conference (APSEC), 2010 17th asia pacific, pp 366–374. IEEE

Tange O (2011) Gnu parallel - the command-line power tool. ;login: The USENIX Magazine 36(1), pp 42–47. http://www.gnu.org/s/parallel

Thung F, Kochhar PS, Lo D (2014) Dupfinder: Integrated tool support for duplicate bug report detection. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14. ACM, New York, pp 871–874. http://doi.acm.org/10.1145/2642937.2648627

Wang S, Lo D, Lawall J (2014) Compositional vector space models for improved bug localization. In: 2014 IEEE international conference on software maintenance and evolution (ICSME), pp 171–180. IEEE

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

White RW, Marchionini G (2007) Examining the effectiveness of real-time query expansion. Inf Process Manag 43(3):685–704. https://doi.org/10.1016/j.ipm.2006.06.005. http://www.sciencedirect.com/science/article/pii/S0306457306000951. Special Issue on Heterogeneous and Distributed IR

Zhang Y, Lo D, Xia X, Sun JL (2015) Multi-factor duplicate question detection in stack overflow. J Comput Sci Technol 30(5):981–997. https://doi.org/10.1007/s11390-015-1576-4

Abram Hindle is an associate professor of Computing Science at the University of Alberta. His research focuses on problems relating to mining software repositories, improving software engineering-oriented information retrieval with contextual information, the impact of software maintenance on software energy consumption (Green Mining), and how software engineering informs computer music. He likes applying machine learning in music, art, and science. Sadly Abram has no taste in music and produces reprehensible sounding noise using his software development abilities. He has published several papers in international conferences and journals, including EMSE, ICSE, FSE, ICSM, MSR, and SANER. He has served on the program committees of several international conferences, and has program co-chaired MSR and SCAM. Abram received a PhD in computer science from the University of Waterloo, and Masters and Bachelors in Computer Science from the University of Victoria. Contact him at abram.hindle@ualberta.ca, http://softwareprocess.ca.

Curtis Onuczko is the Manager of Quality Technology for BioWare, a video-game company that develops high-quality console, PC, and online role-playing games focused on rich stories, unforgettable characters, and vast worlds to discover. He has developed tools for several award-winning video games, such as Mass Effect?, Dragon Age? and Anthem which is currently in development, mainly with a focus on improving quality assurance. He has a passion for software engineering and project management with the goal of improving the software development life cycle. Curtis received a Masters and Bachelors in Computing Science from the University of Alberta. Contact him at curtiso@bioware.com.