Stopping Duplicate Bug Reports before they start with Continuous Querying for Bug Reports

Abram Hindle[1]

[1]**Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, hindle1@ualberta.ca**

ABSTRACT

Bug deduplication 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: continuous 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 continuous querying 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 show some promise for addressing 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 report detection, bug report deduplication

INTRODUCTION

When software stops working the end-user is often asked to report on their experience and describe to the software developers the problem they encountered. 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 can't understand duplicate reports, users couldn't find the duplicate report, etc. Bettenburg et al. (2008); Rakha et al. (2015) End-users do not 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). 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. Most duplicate bug report literature queries by example: they use the whole documents as their search query. 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.

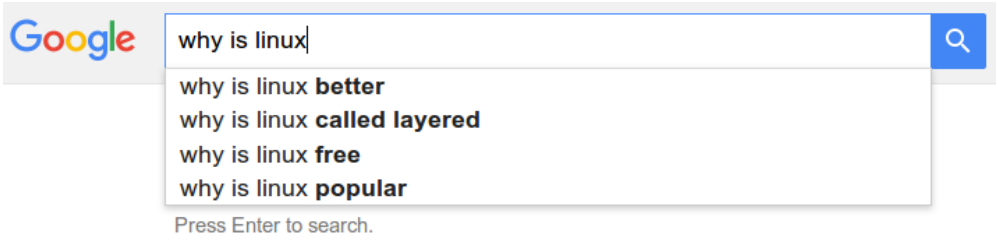

Figure 1. Google Suggest suggesting possible query completions.

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 searched already 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 (in-progress) bug report as numerous queries against the bug tracker. We call this *Continuous Query*. 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 that 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 words of a bug report.

In this paper:

- We propose *Continuous Querying Bug Reports* with *Continuous Query*;
- 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.

PRIOR WORK

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

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 Suggest API and provides possible query suggestions. Figure 1 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. 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. 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. Sun et al. (2011) exploited BM25 and BM25F for query result scoring and ranking.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2373v1 | CC BY 4.0 Open Access | rec: 18 Aug 2016, publ: 18 Aug 2016

Continuous query is a kind of query reformulation as described by Sonia Haiduc Haiduc (2014). Essentially the lack of query reformulation with in issue trackers makes continuous query necessary.

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). Zhang et al. Zhang et al. (2015) have applied typical bug-deduplication technology to StackOverflow duplicate question detection.

Thung et al. 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. Rocha et al. (2015) discuss a similar tool. Rakha et al. 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.

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 away.

Code Recommenders

Recommender systems are a relevant topics to continuous query. Ponzanelli et al. Ponzanelli et al. (2014, 2013) 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 continuous query except far more elaborate, as they do much query preprocessing. Asaduzzaman et al. 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 continuous query is more so the domain, continuous query is initially envisioned for bug reports.

CONTINUOUSLY QUERY BUG REPORTS

Continuously Query Bug Reports, or *Continuous Query* for short, is the attempt to continuously query relevant documents during the creation of a document. The intent of continuous query for 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 continuous query.

Continuous Query is different than classical IR document querying because it asks many queries for 1 document against a system. 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.

Evaluation Methodology

How do we evaluate the effectiveness of a continuous query? 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 of containing more and more words from the duplicate bug report until a maximum threshold has been met. That is 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. The sooner that a duplicate bug report appears the better the performance of the Continuous Query algorithm.

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 (25 words) is met. 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.

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

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2373v1 | CC BY 4.0 Open Access | rec: 18 Aug 2016, publ: 18 Aug 2016

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.

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 LDA or even LSI.

The third context, historical split validation, chooses N pivot bug reports where the that 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.

In this paper we use the third context, historical split validation. We feel that the other contexts can be used for valid evaluations but they might limit reproducability or applicability of the results.

Evaluation Measures

How can we evaluate the effectiveness of a continuous query?

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

Furthermore a continuous query algorithm can only return so many duplicate bug report suggestions at a time. As per search engine IR mythology more than 10 results is often too many. But these are a multitude of queries so only so much can be evaluated, Google Suggestions uses 4 results for 1 query. Many prior works in bug deduplication and triage use top 5 results. 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 continuous query algorithm will return relevant duplicate bug reports sooner, thus higher rank correct results are more valued than lower rank correct results. This implies that treating rank reciprocally or as a weighted factor is important to evaluation.

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.

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.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2373v1 | CC BY 4.0 Open Access | rec: 18 Aug 2016, publ: 18 Aug 2016

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

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

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

MRR, 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. Some prior work uses MRR for bug de-duplicating by suggesting only the first match mattersSun et al. (2011). In continuous query 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 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 continuous query MAP is used to measure 1 bug reports set of continuous queries. When evaluated 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.

These measures will determine if we find our duplicates quickly or not as those queries who take too many words to find duplicate bug reports will start off with low scores.

AveP-TOPk is a combination of MAP and TOPk. AveP-TOPk first takes the continuous queries and ranks them by TOPk, thus each continuous query is evaluated as 1.0 or 0.0 depending if a value 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 continuous query algorithms who return duplicates in the fewest number of words. Imagine a continuous query of 5 words for "at the small dog house", where all 5 word 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 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

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

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2373v1 | CC BY 4.0 Open Access | rec: 18 Aug 2016, publ: 18 Aug 2016

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

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

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 makes more sense as it weights continuous query algorithms who return good TOP5 results sooner more than those who take longer to produce TOP5 results.

EXPERIMENT METHODOLOGY

Thus we want to demonstrate continuous query's naive performance to show that it can indeed work, that it is indeed feasible to stop duplicate bug reports before they start.

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

- RQ1: How well does our implementation of continuous query work?
- RQ2: What percentage of duplicate bug reports could be stopped before they start?
- RQ3: Does stemming of tokens matter with respect to continuous query?

The validation method we use is the third method we described in Section 3.1: historical split validation. Historical split validation is used to be fair to future implementations of continuous query, which might use models that are slow to train (e.g., an LDA-based continuous query model 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 3.1 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 continuous query 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 of the bug report title concatenated with the bug report description. The documents queried can be the concatenation of bug report titles, bug report description, and bug report comments. In this paper we concatenate titles and description, but depending on the model used training of a model can be executed differently.

Experimental Naive Continuous Query Model

In this paper we provide a naive version of continuous query. We expect there are more intelligent ways to implement continuous query that perform better in terms of run-time and information retrieval performance. Our naive continuous query system uses an information retrieval model consisting of TF-IDF transformation of documents, with or without Porter Stemming, with stop word removal, with digits

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2373v1 | CC BY 4.0 Open Access | rec: 18 Aug 2016, publ: 18 Aug 2016

Project	Source	# Bug Reports	# Dupes	# Dupe Buck-ets	Mean Title	Mean De-scription	Earliest	Latest
Android	GoogleCode	37626	1363	788	10.447	182.789	2007-11-12	2012-09-19
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
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
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
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

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

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 an 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.

Datasets

In this paper we use 12 different issue tracker repositories. These datasets are described in Table 1 The dataset used in this paper is available at: `https://archive.org/details/2016-04-09ContinuousQuer` The code to run the experiments on this data is available at: `https://bitbucket.org/abram/continuous-q` 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, as 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, 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 app that manages email. Mozilla is a company who's flagship product is Firefox. 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 continuous query.

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 of each of the 12 evaluated projects. We evaluated continuous queries of duplicate bug reports on these models and aggregated the results for discussion in this section.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2373v1 | CC BY 4.0 Open Access | rec: 18 Aug 2016, publ: 18 Aug 2016

Project	Exp	Old	Continuous Query					
		MAP	MAP	MRR	TOP1	TOP5	TOP10	aveP-TOP5
Android	Stemming	0.240	0.231	0.239	0.183	0.278	0.368	0.241
Android	NoStem	0.214	0.200	0.208	0.135	0.286	0.374	0.261
App Inventor	Stemming	0.278	0.310	0.413	0.321	0.539	0.575	0.490
App Inventor	NoStem	0.308	0.291	0.390	0.310	0.535	0.575	0.476
Bazaar	Stemming	0.287	0.200	0.200	0.118	0.276	0.354	0.236
Bazaar	NoStem	0.295	0.202	0.202	0.127	0.271	0.349	0.236
Cyanogenmod	Stemming	0.475	0.202	0.208	0.149	0.263	0.326	0.221
Cyanogenmod	NoStem	0.343	0.130	0.137	0.098	0.139	0.232	0.129
Eclipse	Stemming	0.334	0.201	0.203	0.141	0.262	0.329	0.225
Eclipse	NoStem	0.344	0.191	0.192	0.129	0.251	0.318	0.221
K9Mail	Stemming	0.342	0.340	0.343	0.192	0.540	0.601	0.454
K9Mail	NoStem	0.311	0.317	0.320	0.203	0.500	0.615	0.414
Mozilla	Stemming	0.316	0.205	0.209	0.149	0.263	0.338	0.219
Mozilla	NoStem	0.306	0.202	0.205	0.143	0.258	0.327	0.215
MyTrack	Stemming	0.179	0.446	0.454	0.363	0.565	0.580	0.482
MyTrack	NoStem	0.173	0.415	0.422	0.345	0.500	0.523	0.420
OpenOffice	Stemming	0.312	0.203	0.205	0.151	0.246	0.313	0.195
OpenOffice	NoStem	0.297	0.178	0.181	0.130	0.223	0.300	0.174
OpenStack	Stemming	0.454	0.263	0.263	0.165	0.404	0.489	0.329
OpenStack	NoStem	0.452	0.270	0.270	0.175	0.415	0.494	0.342
osmand	Stemming	0.128	0.127	0.127	0.051	0.153	0.318	0.121
osmand	NoStem	0.151	0.144	0.144	0.049	0.160	0.447	0.152
Tempest	Stemming	0.307	0.266	0.266	0.192	0.327	0.445	0.322
Tempest	NoStem	0.320	0.277	0.277	0.207	0.340	0.478	0.331
Average	Stemming	0.304	0.249	0.261	0.181	0.343	0.420	0.294
Average	NoStem	0.293	0.235	0.246	0.171	0.323	0.419	0.281
Average	Both	0.299	0.242	0.253	0.176	0.333	0.419	0.288

Table 2. Performance per Project with Continuous Query. Old MAP is not Continuous Query.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2373v1 | CC BY 4.0 Open Access | rec: 18 Aug 2016, publ: 18 Aug 2016

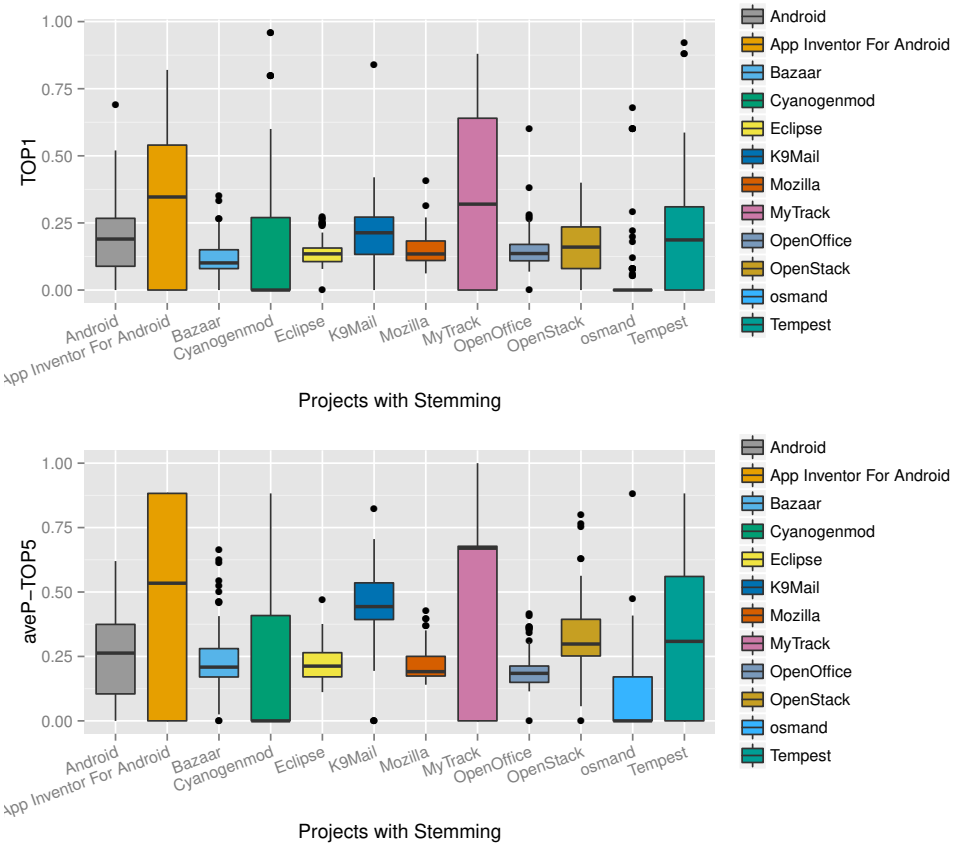

Figure 2. Boxplots of the distributions of evaluation measures for all 12 projects.

RQ1: How well does our implementation of continuous query work?

The performance of continuous query 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 continuous query 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 except TOP5, 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 continuous query results. TOP5 had a p-value of 0.049 which is far too close to the α. Thus continuous query does not have as good performance as 1 single query, but it still have enough performance that it can be used to prevent duplicate bug reports. Figure 4 summarizes graphically the difference between entire document querying and the continuous querying 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 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 Figure 3 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 – it takes time to make duplicate bug reports.

The Android results in Figure 3 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.

What we learn from RQ1 is that typically the performance of continuous query in the first 25 words

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2373v1 | CC BY 4.0 Open Access | rec: 18 Aug 2016, publ: 18 Aug 2016

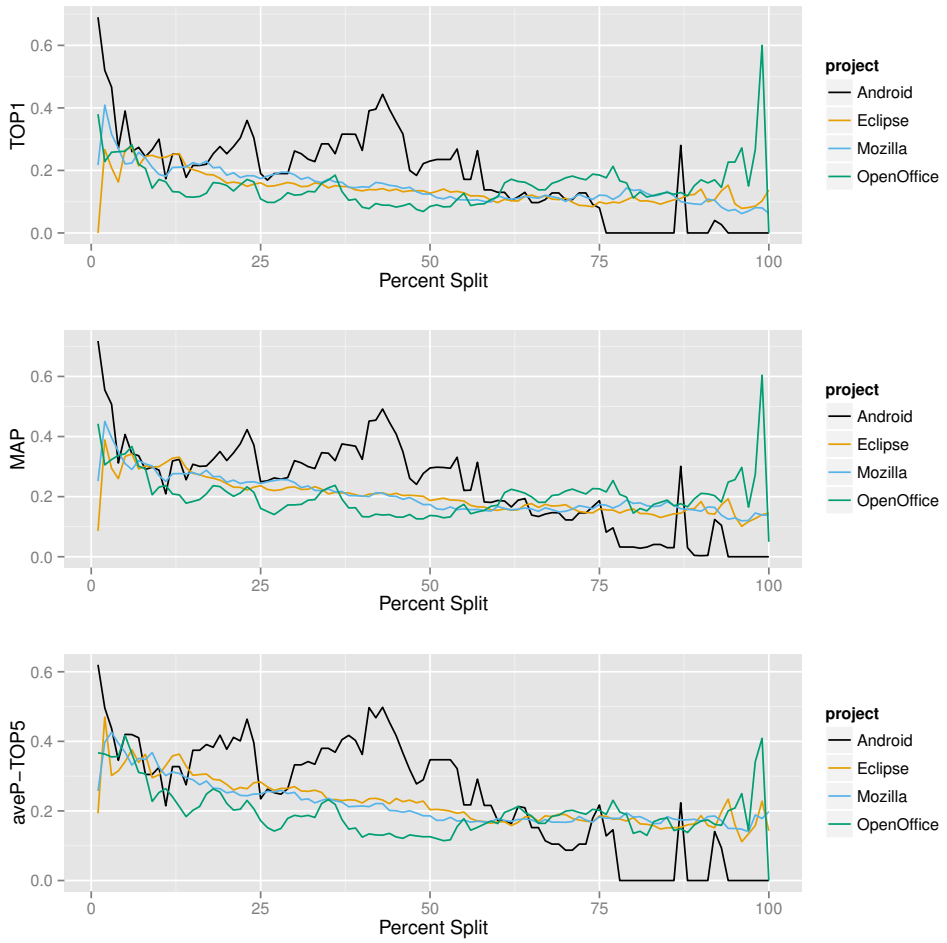

Figure 3. Evaluation metrics, TOP1, MAP, aveP-TOP5 performance over 100 splits across 4 different projects.

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 report.

RQ2: What percentage of duplicate bug reports could be stopped before they start?
Based on 2 we can see that TOP1 across projects is 0.17 to 0.18. 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 0.32 and 0.34. 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 around 14 to 15 words on average are needed to find a duplicate bug report ($AveP-TOP5(q) = (1/15+2/16+...+10/24+11/25)/11 = 0.281$). But this is actually a skewed result. In projects like OpenOffice approximately 62% of the duplicate bug queries do not find the duplicate in any of the continuous query the Top5. Per project these failed continuous queries make up 6% to 62%. That is 38% to 94% of continuous queries (44% in total across all projects), depending on the project, will produce duplicate bug reports in their TOP5 query results. 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 a 38-44% or better reduction in duplicate bugs could be expected for projects that initially implement continuous query. Once continuous query is adopted these statistics would change due to changing behaviours of users and developers.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2373v1 | CC BY 4.0 Open Access | rec: 18 Aug 2016, publ: 18 Aug 2016

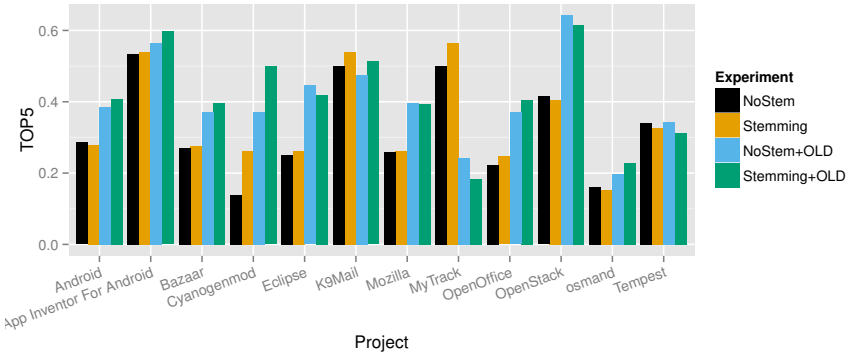

Figure 4. TOP5 score calculated from the entire document, compared against TOP5 score calculated from continuous query.

RQ3: Does stemming of tokens matter with respect to continuous query?

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 5 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.

While the plots make it seem like stemming matters, when we compare the 100 MAP, MRR, TOP1, TOP5, aveP-TOP5 values from each of the stemmed and not-stemmed runs per project we find very few results that are statistically significant when $\alpha \leq 0.05$ when using the paired Wilcoxon rank sum test. For MAP Bazaar, Mozilla, K9Mail, and OpenStack had p-values > 0.05 and thus showed little response to stemming. For aveP-TOP5 Bazaar, Mozilla, Eclipse, Osmand, and Tempest had p-values > 0.05 and thus showed little response to stemming. Thus 7 unique projects indicated insignificant response to stemming. While the rest of the projects had significant p-values indicating there was a difference between stemming and not stemming. Thus we conclude that stemming sometimes has an effect on performance but it is not consistently statistically significant. This echos the work of Panichella et al. Panichella et al. (2016) who showed that optimal IR configurations differ across projects.

LESSONS LEARNED

How would continuous query operate in actual software development? Can we design systems capable of handling the sheer number of queries that continuous querying of bug reports calls for?

We have deployed continuous query through our Bug-Party/D upeBuster product [2] and what we have found is that providing a RESTful webservice that can respond to continuous bug report queries is quite possible with modern NLP indexing. While the prototype implementation depicted in the paper is not the same as this software (Bug-Party uses elasticsearch's TF-IDF implementation), we integrated Bug-Party with existing bug trackers to enable continuous query. By injecting some javascript into a bug tracker's webpage one can add functionality to enable text input fields to continuously query the continuous query webservice.

From the researcher's perspective, what we have learned is that keeping the webservice separate from the bug tracker is important, as we do not want to jeopardize the performance of the bug tracker software. Thus keeping a mirror of the bug tracker bug reports and querying that mirror is safer and more efficient.

The ability to scale continuous query to many clients comes from the efficiency of modern open-source search engines and the caching capabilities of HTTP and the speed of human typing. Client side smart behaviour can avoid querying the continuous query server too much, but careful use HTTP caching headers enables browsers to respond immediately to the user if they type a query that is already cached. Furthermore because HTTP enables caching meta-data one can put a cache in front of the continuous query webservice to improve performance. Continuous query is cache-able due to smart routes, that are repeatable and that the queries are idempotent or nearly idempotent (temporarily cache-able). Thus by providing a separating continuous query webservice that mirrors the bug tracker bug reports and

[2] https://bitbucket.org/abram/bugparty/

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2373v1 | CC BY 4.0 Open Access | rec: 18 Aug 2016, publ: 18 Aug 2016

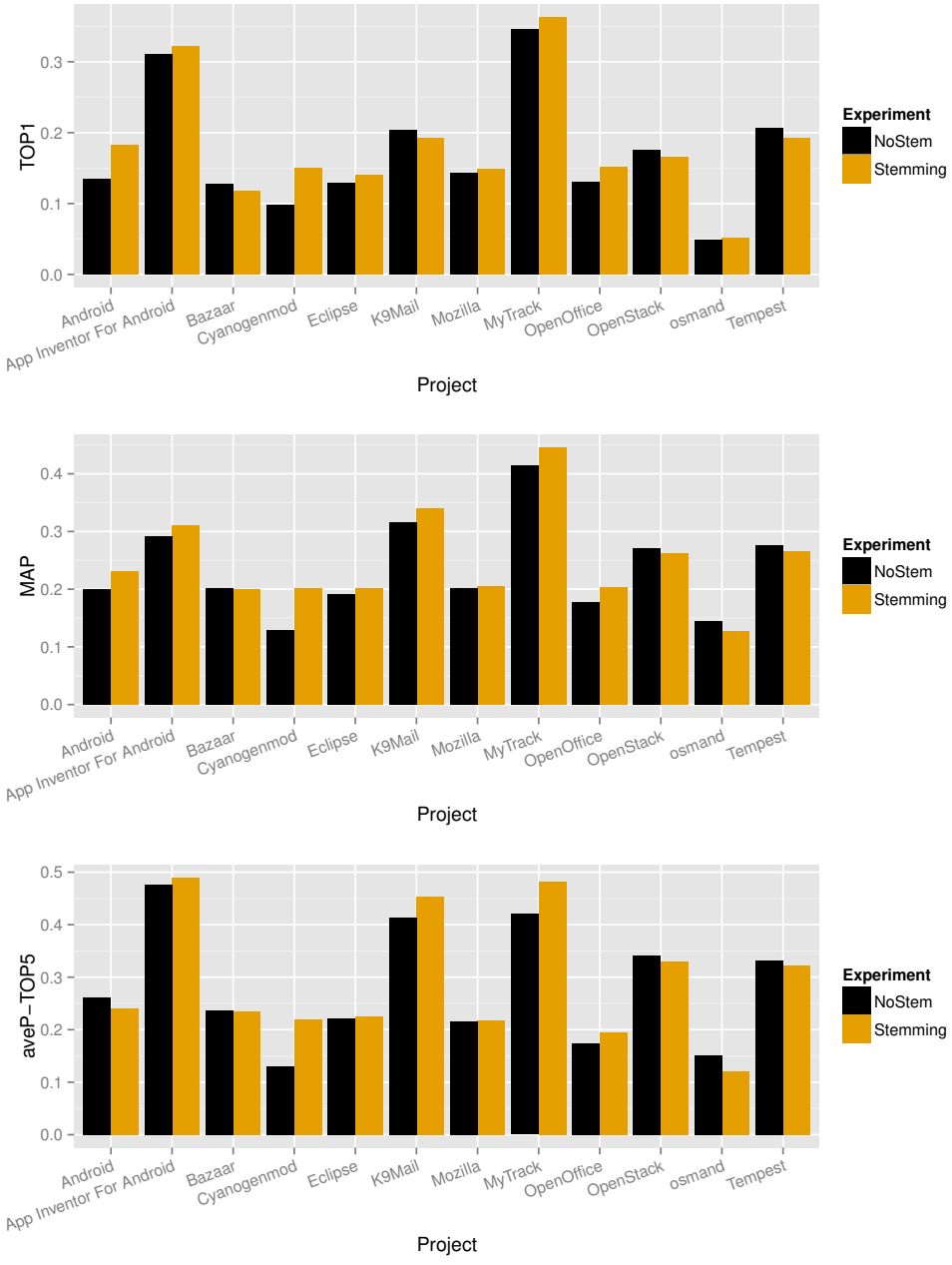

Figure 5. Average performance of stemming versus not stemming across projects

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2373v1 | CC BY 4.0 Open Access | rec: 18 Aug 2016, publ: 18 Aug 2016

takes advantage of HTTP caching capabilities one can provide continuous query service to many users efficiently.

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.

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 and no evidence that this is the case. We also assume that users will read some of the duplicate bug report suggestions yet we cannot present evidence of this yet.

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.

External validity is threatened by the small number of systems and small number of dupes 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 use opportunistically rather than statistically sampled and this will affect the reliability of the results as well as external validity.

CONCLUSION

The Continuous Query 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 continuous 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.

Continuous querying of bug reports is a new kind of bug deduplication task that has the potential to prevent duplicate before they occur in 44% or more of observed duplicate bug report cases. We demonstrated via a rigorous experiment that continuous query is effective at finding duplicate bug reports in multiple systems. 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 continuous query techniques? As this is a new kind of bug deduplication which needs further research into appropriate IR techniques to retrieve bug reports, we have shared our dataset and source code. [3]

Future Work

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

We employed very naive IR in this paper to implement continuous query. We used TF-IDF with cosine distance in this work, where as other bug deduplication works have used BM25 and BM25F Sun et al. (2011, 2010).

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. Continuous Query research can be bolstered with search based software engineering Harman et al. (2012), IR search techniques employed by Panichella et al. Panichella et al. (2016), and query quality and reformulation techniques proposed by Haiduc et al. Haiduc (2014).

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

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2373v1 | CC BY 4.0 Open Access | rec: 18 Aug 2016, publ: 18 Aug 2016

Furthermore the elephant in the room of continuous query 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 continuous query performance.

ACKNOWLEDGMENTS

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

REFERENCES

Aggarwal, K., Rutgers, T., Timbers, F., Hindle, A., Greiner, R., and Stroulia, E. (2015). Detecting duplicate bug reports with software engineering domain knowledge. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 211–220. IEEE.

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

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

Asaduzzaman, M., Roy, C. K., Schneider, K. A., and Hou, D. (2014). Cscc: Simple, efficient, context sensitive code completion. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 71–80. IEEE.

Bettenburg, N., Premraj, R., Zimmermann, T., and Kim, S. (2008). Duplicate bug reports considered harmful... really? In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 337–345. IEEE.

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*, pages 657–662. IEEE Computer Society.

Harman, M., Mansouri, S. A., and Zhang, Y. (2012). Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–11:61.

Jalbert, N. and Weimer, W. (2008). Automated duplicate detection for bug tracking systems. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 52–61. IEEE.

Klein, N., Corley, C. S., and Kraft, N. A. (2014). New features for duplicate bug detection. In *MSR*, pages 324–327.

Lazar, A., Ritchey, S., and 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*, pages 308–311. ACM.

Panichella, A., Dit, B., Oliveto, R., Penta, M. D., Poshyvanyk, D., and Lucia, A. D. (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*, pages 314–325. IEEE Computer Society.

Ponzanelli, L., Bacchelli, A., and Lanza, M. (2013). Seahawk: Stack overflow in the ide. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1295–1298, Piscataway, NJ, USA. IEEE Press.

Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R., and 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, pages 102–111, New York, NY, USA. ACM.

Rakha, M. S., Shang, W., and Hassan, A. E. (2015). Studying the needed effort for identifying duplicate issues. *Empirical Software Engineering*, pages 1–30.

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

Rocha, H., De Oliveira, G., Marques-Neto, H., and Valente, M. T. (2015). Nextbug: a bugzilla extension

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2373v1 | CC BY 4.0 Open Access | rec: 18 Aug 2016, publ: 18 Aug 2016

for recommending similar bugs. *Journal of Software Engineering Research and Development*, 3(1):1–14.

Runeson, P., Alexandersson, M., and Nyholm, O. (2007). Detection of duplicate defect reports using natural language processing. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 499–510. IEEE.

Sun, C., Lo, D., Khoo, S.-C., and 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*, pages 253–262. IEEE Computer Society.

Sun, C., Lo, D., Wang, X., Jiang, J., and Khoo, S.-C. (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*, pages 45–54. ACM.

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

Thung, F., Kochhar, P. S., and 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, pages 871–874, New York, NY, USA. ACM.

Wang, X., Zhang, L., Xie, T., Anvik, J., and 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*, pages 461–470. ACM.

Zhang, Y., Lo, D., Xia, X., and Sun, J.-L. (2015). Multi-factor duplicate question detection in stack overflow. *Journal of Computer Science and Technology*, 30(5):981–997.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2373v1 | CC BY 4.0 Open Access | rec: 18 Aug 2016, publ: 18 Aug 2016