Search-Based Duplicate Defect Detection: An Industrial Experience

Mehdi Amoui, Nilam Kaushik, Abraham Al-Dabbagh, Ladan Tahvildari
University of Waterloo
Waterloo, Canada
{mamouika, nkaushik, ataldabb, ltahvild}@uwaterloo.ca

Shimin Li, Weining Liu
BlackBerry Inc.
Waterloo, Canada
{shili, wliu}@rim.com

***Abstract*—Duplicate defects put extra overheads on software organizations, as the cost and effort of managing duplicate defects are mainly redundant. Due to the use of natural language and various ways to describe a defect, it is usually hard to investigate duplicate defects automatically. This problem is more severe in large software organizations with huge defect repositories and massive number of defect reporters. Ideally, an efficient tool should prevent duplicate reports from reaching developers by automatically detecting and/or filtering duplicates. It also should be able to offer defect triagers a list of top-N similar bug reports and allow them to compare the similarity of incoming bug reports with the suggested duplicates. This demand has motivated us to design and develop a search-based duplicate bug detection framework at BlackBerry. The approach follows a generalized process model to evaluate and tune the performance of the system in a systematic way. We have applied the framework on software projects at BlackBerry, in addition to the Mozilla defect repository. The experimental results exhibit the performance of the developed framework and highlight the high impact of parameter tuning on its performance.**

***Index Terms*—Duplicate Defect Detection, Parameter Tuning, Search-based Software Engineering, Information Retrieval**

I. Introduction

A defect can be reported at any stage during software development. In large-scale software projects searching the defect tracking system to determine whether a problem has already been reported is usually higher than the cost of creating a new defect report [1]. For example, Cavalcanti *et al.* analyzed 8 open source projects and found that the average time spent on searching and analyzing defect reports was 12.5 minutes, and an average of 48-person hours were spent per day on searching duplicate defect reports [2].

Due to the associated costs of hidden duplicate defects, one of the main tasks of the triaging process is to determine whether a defect report is a duplicate of an existing defect as early as possible. However, preventing the introduction of duplicate defect in the first place is a costly process and it might be impossible in some cases. In addition, correctly marked duplicate defect reports are not necessarily harmful, as they may provide additional information for developers to resolve a defect [3].

In order to reduce the cost of managing duplicate defects, effective duplicate detection is one of the desired features for next generation defect tracking systems [4]. Ideally, a defect tracking system can prevent users from reporting duplicate defects with minimal overhead. For an incoming defect report, the system should be able to detect and/or filter duplicates and hence offer the defect triagers a list of top-N similar reports from the defect (bug) repository. This allows triagers to compare the similarity of the incoming defect report with the suggested duplicates.

BlackBerry, a leading designer, manufacturer and marketer of innovative wireless solutions for the worldwide mobile communications market, is developing and maintaining many interconnected large-scale software systems. Having an efficient duplicate defect detection system can have a great impact on the efficiency of such an enormous software ecosystem. This demand motivates us to design and develop an effective duplicate defect detection system at BlackBerry that can: i) save triagers' time to find related defect reports, and ii) help developers gather more information about the defect at hand, in turn reducing the time to resolve a defect.

Several techniques have been proposed to help triagers with the process of duplicate defect detection (e.g., [1], [5], [6], [7], [8]). These approaches are either to prevent duplicate reports from reaching developers by automatically filtering them (e.g., [6]), or to suggest a list of similar defect reports, allowing triagers to compare incoming defect reports with the suggested list (e.g., [5]). The performance of these approaches has mainly been evaluated with the datasets of open-source projects.

In this paper, we present an industrial practice in duplicate defect detection for large scale software systems. We have built and tuned a search-based Duplicate Defect Detection (DDD) framework at BlackBerry. The framework has two main use cases: i) when a reporter is logging a new defect, it provides the option to search for a similar and existing duplicate defects, and ii) it provides triagers the ability to check for duplicates of an existing defect. The approach follows a generalized process model to evaluate and tune the performance of the system. We apply the framework on one of the main defect repositories of BlackBerry. Additionally, we study the impact of parameter tuning on the defect repository of the Mozilla open-source project. The experimental results exhibit the performance of the developed tool on a large-scale industrial software project and highlight the high impact of parameter tuning on the system's performance. It is notable to mention that DDD is currently operational as a service and it is being used to support BlackBerry's defect management process.

978-1-4673-2936-1/13 © 2013 IEEE

MSR 2013, San Francisco, CA, USA

This paper is organized as follows: Section II outlines our search-based duplicate-defect detection approach and gives an overview of the technologies that are involved in our implementation. Section III details our experimental setup and presents the obtained results. Section IV presents a discussion on the results. Related work is covered in Section V. Finally, we conclude this paper and outline future work in Section VI.

II. The Approach

The domain and characteristics of a software project need to be considered for effective and high quality duplicate defect detection [9]. In view of this, we need a feedback system to assess the search quality and tune parameters for each software product. For this purpose, we have developed the Duplicate Defect Detection (DDD) framework. A high-level view of the framework is illustrated in Figure 1. DDD is composed of a customized search-engine and a feedback loop for evaluating and tuning the search quality for each project and its user requirements. In this section, we first describe the search process and then we elaborate on the parameter tuning and evaluation process of this framework.

A. Search-Based Duplicate Defect Detection

There are a number of search-based techniques used to solve various software engineering problems [10]. Duplicate defect detection can be also modeled as a search problem. Each defect report, including all its relevant fields and data, can be modeled as a textual document. The search problem is to find all documents that are highly similar or identical to other documents. However, as there is no unique way of describing a defect, searching for duplicate-defects is a challenging and error-prone procedure.

Using information retrieval techniques for full-text indexing, processing, and searching is a promising approach towards duplicate defect detection [9]. There are a number of full-text search-engines available that can be customized and used for mining duplicate (similar) defects. Among them, our approach is based on Apache Lucene, a popular open-source full-text search engine library with a wide application foreground [11]. Using Lucene to search for similar documents is a three step process: i) preparing the data schema and indexing the data using the provided API, ii) setting search parameters and querying the system, and iii) refining and presenting the results. A more detailed description of each step in the context of duplicate defect detection is as follows:

1) Pre-processing and Indexing: The first step in full-text search is data pre-processing and indexing. Pre-processing usually includes: cleaning raw data and removing irrelevant data, keyword injection, merging and splitting fields, and many other data preparation actions [12]. After pre-processing, the data is ready to be indexed. In this step, relevant fields extracted from defect reports are indexed with variable boost factors based on their importance. The index is created based on a user-defined schema, which contains information about the valid fields in an index and the types of those fields. The index can also include a term-vector (i.e., a collection of term-frequency pairs) for each data field, which will be used by the ranking algorithms to measure document similarities.

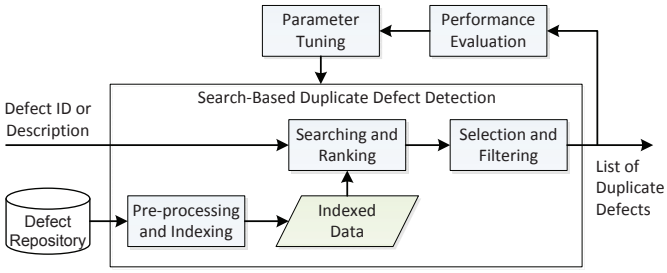

Fig. 1. The Duplicate Defect Detection (DDD) Framework

2) Searching and Ranking: In DDD, searching for duplicate defects is a full-text search on defect reports (documents) to find the matching pairs. Document matching is the process of calculating similarity between documents and ranking (sorting) the result set. Lucene features a *"More Like This (MLT)"* functionality for finding documents similar in content to the query document. To detect duplicate defects, an MLT query is performed to compare the queried defect document with all other defects in the repository. Accompanying the MLT functionality are parameters that can be fine tuned based on the dataset that is being queried.

Ranking is the process of ordering the result set. In Lucene a customizable sort function orders the result set by their calculated similarity value (a.k.a. *score*) in descending order. The score is calculated by the used ranking algorithm. However, as the sort function is customizable, other factors and parameters can be also included to reorder the result set as needed. For example, the scoring function can be a weighted product of score and defect report recency, which means that more recent defects will tend to appear higher in the result set.

3) Selection and Filtering: Selection is the process of finalizing the result set that will be presented to the user. The main goal of selection is to reduce the number of false positives in the result set. False positives reduce precision and increase the effort required by users to investigate the result set for true duplicate defects.

We can reduce the number of false positives by setting a cutoff point. For example, the cutoff can be set to 1 result, which means only the top result will be reported as a duplicate. This will result in a hit or miss situation (either 100% recall and 100% precision, or 0% recall and 0% precision) which can only be considered as a practical solution if there exists only one duplicate defect for a given input defect and the performance of similarity and ranking algorithms is extremely high for the working dataset. A more practical solution, as also suggested in [5], is to report a set of potential duplicates (top-N results) based on the hypothesis that true duplicates have more similarity to the input defect and hence are ranked higher in (among top-N) results.

When presenting the search results, we can filter the results by their similarity score. Before doing so, we need to normalize the scores of all results by dividing them against the first

search result (the highest scoring). With each result having a normalized similarity score, a cutoff is formed based on a chosen minimum score threshold between 0 and 1. However, each dataset has its own characteristics with a specific ratio of duplicates defects to total defects and having a fix cutoff value will result in sub-optimal performance as the precision may have a room for improvement. Therefore, the cutoff value is one of the key parameters to be tuned for each data set in the DDD framework.

B. Performance Evaluation

Many different measures are available for evaluating the performance of information retrieval systems [12]. Most of these measure (e.g., F-measure) are based on recall and precision. Recall is the fraction of relevant instances (i.e., true duplicates) that are retrieved, and precision is the fraction of retrieved instances that are relevant [13]. Recall indicates the performance of the 'search' component, and precision represents the performance of the 'selection' component. The total recall rate is measured as the proportion of defects in the list of duplicate defect pairs which returned their respective true duplicates within the search results. That is to say, if we have a set of 100 duplicate defect pairs, and running the duplicate detector on all of them results in 50 defects that return their true duplicate, we have a 50% recall rate. For the measurement of precision, the number of true positives (true duplicates) found is compared against the total number of search results across all queries. That is to say, if 100 queries return 1 result each, and 50 of them contain the true duplicate in that result, then we have a precision of 50%.

To evaluate the performance of the framework, we generate a list of duplicate defect pairs. A defect can be marked as duplicate of another by triagers or testers; this linkage is what we extract from the defect repository to generate each true duplicate defect pair, or in other words the ground truth. Using this list, we set out to run a duplicate defect search against every element in the list of duplicate pairs. Having the ground truth, every defect in the resultset can be marked to be either relevant or non-relevant to a particular query.

C. Parameter Tuning

Parameter tuning is the the process of finding and setting an optimal value of the parameters that determine a system's characteristics. The process can be specified by the parameters and the tuning procedure. Parameter tuning is an offline procedure in which parameter values remain fixed during the run, and should be distinguished from parameter control in which initial parameter values are changed during the run [14]. Each sub-process in DDD has a set of parameters that impact the performance of duplicate defect detection. In general, we can identify four classes of parameters to tune each step of the search-based duplicate detection process as follows:

- ***Data:*** The choice of correct preprocessors and tuning the indexed data is an important factor in full-text search. The parameters of this class are those that affect the indexed data. This includes wide range of pre-processors, which can be applied on the dataset (e.g., data pruning, keyword injection), and the ones that directly affect the indexed data (e.g., merging and splitting data fields).
- ***Search:*** This parameter set includes anything that affects search results on the indexed data. Including but not limited to the query itself, and incorporation of other data and information (e.g., list of stopwords and synonyms) that affect search results. The choice of the underlying retrieval algorithm can also impact results as shown in [9].
- ***Algorithm:*** This class includes search algorithm specific parameters. Lucene supports four ranking algorithms out of the box: Term Frequency-Inverse Document Frequency (TF-IDF), BM25F (an extension of the popular Okapi BM25 algorithm), Jelinek-Mercer smoothing, and Dirichlet smoothing [12]. However, the extensible architecture of Lucene allows the addition of customized and additional search algorithms (e.g., an extension of BM25F proposed by Sun et al. [15]).
- ***Selection:*** There are a number of ways to select and present duplicate defects (e.g., selecting top-N results). Custom sort functions can be defined to optimize the search quality by reordering the result set. Moreover, customizing the presentation of the results could enhance the user-experience (e.g., clustering the results).

The (near-)optimal value for each parameter can vary based on the defect repository and/or system requirements. As we will present in our experimental study, some parameters have a great impact on the total performance of the system. However, the abundant combinations of parameters that need to be tested out to achieve desirable results calls for the use of heuristics. In parameter tuning, parameters to be tuned are usually modeled as vectors. The goal of the tuning procedure is to find a vector with maximum utility, where utility is based on some definition of system performance and some objective functions (requirements) [16]. Finding optimal parameters through exhaustive search can be very time-consuming (if not impossible) as the search space grows exponentially with the number of parameters to be tuned and number of valid values for each of them. For example, considering six parameters and five values for each of them, one has to test 5^6 = 15625 different setups. This implies that trying all different permutations is practically impossible, which necessitates search-based optimization.

Our selected approach for this study is a class of one-dimensional optimization techniques, known as Line Search methods [17]. In Line Search, from an initial (random) point in the parameter space, a search is performed for one parameter at a time while fixing the values of the other dimensions. This basic procedure is repeated for each of the tuning parameters. The optimal values for each dimension determine a new optimal point in the parameter vector space. A line between the original point and the new computed point is considered as a promising direction. Next, a number of parameter sets on this line will be selected, and the procedure is repeated to find a new direction. The detailed procedure and the application of line search in parameter tuning is described in [18].

TABLE I
TUNING PARAMETER LIST

Class	Name	Default Value	Valid Range
Data	Data Field Combination	All	$\binom{All}{field}$
	Field Boost Factor	1.0	[0.0 - 1.0]
Search	Stopwords	False	{true, false}
	Synonyms	False	{true, false}
	TimeFrame	N/A	[1 - 90] days
	Max. Doc. Freq.	N/A	[1 - 100]%
	Algorithm	TF-IDF	All Choices
Algorithm	λ : Jelinek Mercer	N/A	[0.1 - 0.9]
	k1 : BM25F	1.2	[1.2 - 2.0]
	b : BM25F	0.75	[0.5 - 1.0]
	μ : Dirichlet	2000	[0 - 10000]
Selection	Score Cutoff Threshold	N/A	[1 - 500]

III. EXPERIMENTAL STUDY

As a proof of concept and to study the performance of our approach within the BlackBerry domain, we performed an experimental study on one of the main and highly active defect repositories at BlackBerry. Additionally, a study on the defect repository of the Mozilla project is performed in order to validate the reusability of our approach, which can also give an insight on the performance of our approach on a long-lived open-source project.

The main focus this experimental study is designated to present the performance of the developed framework and to highlight the impact of parameter tuning on the system's performance. Since the cost of missing duplicate defects in the search results is significantly higher than cost of false positives at BlackBerry, we use recall rate to evaluate the performance and select and tune a subset of possible parameters to maximize recall rate. The complete list of these parameters including their valid ranges is shown in Table I.

Throughout this section, we will present analysis steps, design decision, and detailed results of applying our approach on the BlackBerry dataset, followed by a summary of applying DDD on the Mozilla dataset.

A. BlackBerry Datasets

The selected dataset has 307,437 defects with 7,513 duplicate defect pairs. The dataset consists of open and closed defects from multiple BlackBerry production software projects over a 3 years time span. The majority of the defects in this dataset are reported by BlackBerry employees. There are processes in place to review every new defect by triage teams and then by domain experts to identify duplicates. Once a duplicate is identified, the duplicate pairs are linked manually in the defect tracking system. The majority of defects have only one duplicate pair. However, it is possible that a defect has more than one duplicate. For our studied dataset, the number of duplicates per defect is shown in Figure 2. This statistic suggests that binary matching (hit or miss) is not the best solution for BlackBerry, as identifying and analyzing multiple duplicates of the same defect or those that are quite similar to it provides insightful information for triagers.

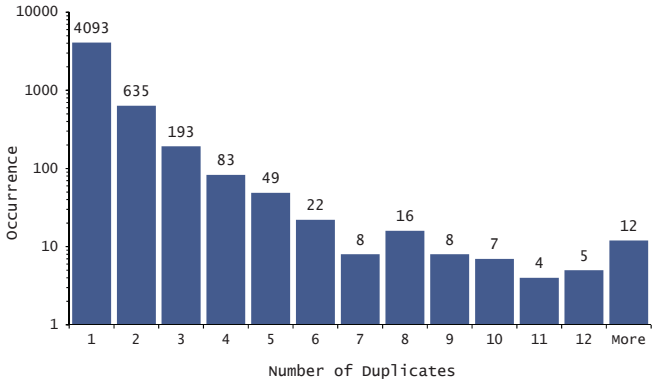

Fig. 2. Number of Duplicates per Defect

```
<defect>
  <defect_id>257966</defect_id>
  <creation_date> 2008-12-08 13:58:00 -0500</creation_date>
  <summary>NLSKeyHyperlink leaks type binding via
   AccessorClassReference</summary>
  <description>Traceback: org.eclipse.jdt.internal.ui.
   javaeditor.NLSKeyHyperlink [0] of org.eclipse.jface.text.
   hyperlink.IHyperlink[2] fInformation of org.eclipse.jface
   .text.hyperlink.MultipleHyperlinkPresenter
   $MultipleHyperlinkHoverManager fManager of org.eclipse.
   jface.text.hyperlink.MultipleHyperlinkPresenter
   fHyperlinkPresenter of org.eclipse.jdt.internal.ui.
   javaeditor.CompilationUnitEditor$AdaptedSourceViewer
   fViewer of org.eclipse.jdt.internal.ui.text.
   JavaReconciler this$0 of org.eclipse.jface.text.
   reconciler.AbstractReconciler$BackgroundThread [JNI
   Global, Stack Local, Thread]</description>
  <classification>Eclipse</classification>
  <product>JDT</product>
  <component>Text</component>
  <version>3.5</version>
  <defect_status>RESOLVED</defect_status>
  <resolution>FIXED</resolution>
  <reporter name="Person X">Person_X</reporter>
  <assigned_to name="Person Y">Person_Y</assigned_to>
  <First_Email>HEAD NLSKeyHyperlink leaks type binding via
   AccessorClassReference. I had an editor open on
   ReorgPolicyFactory, and in a memory dump, I found a
   NLSKeyHyperlink which retained about 10MB of memory, all
   via AccessorClassReference.fBinding. I verified that the
   type binding was not referenced from anywhere else. I
   guess the memory would eventually be freed when I closed
   the editor or maybe when I opened another link, but 10MB
   is still too much to give away.
  Observed Result:
  I had an editor open on ReorgPolicyFactory, and in a
   memory dump, I found a NLSKeyHyperlink which retained
   about 10MB of memory.
  Steps To Reproduce:
  N/A
  </First_Email>
</defect>
```

Listing 1. A Sample Defect Report in XML Format.

Listing 1 shows a sample defect report in XML format. The notable fields that the report contains include the summary, description, and First_Email. The summary field is also commonly referred to as the title of the bug or the short description. The description field contains all of what the reporter perceives as valuable information relating to the bug. The First_Email field is a record of the email conversation between the defect reporter and the product support and developers before the defect was reported. The First_Email

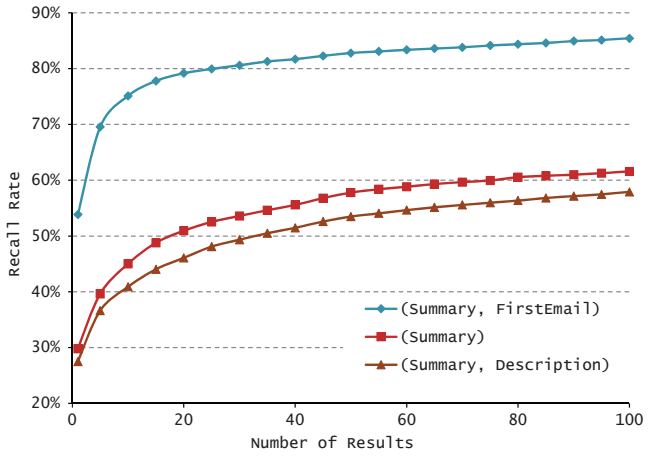

Fig. 3. Recall Rates for Different Field Combinations

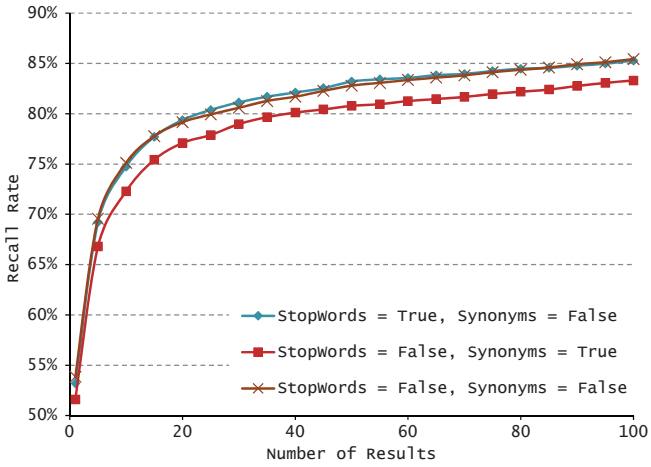

Fig. 4. Recall Rates with Stopwords and Synonyms

field is relevant mainly because it consists of the subfields Observed_Result and Steps_To_Reproduce. Each field has a Field Length, which denotes the average number of terms in that field across all documents. Terms themselves are used for assessing the similarity of one item against another.

B. Data Field Parameters

Within the defect repository of our dataset, a defect report consists of three significant fields: Summary, Description and First_Email, as shown in Table II. When examining a long field such as First_Email, with on average over 100 terms, the task of parsing what is perceived as relevant information can prove to be lucrative. In our attempts to do so, we set out to extract two main components from the First_Email: Observed Results and Steps to Reproduce. The supposed benefits to the reduction of the First_Email field include a potential reduction of noise to the system as well as increased indexing performance. We will see later how this clipping of the First_Email field affects the performance.

We evaluate the search quality for different combinations of the specified fields with default TF-IDF algorithm. As shown in Figure 3 for the defect repository, the Description field seemed to only generate noise, since searching Summary on its own turned out to yield better recall results. With all combinations tested, we found the best results for the dataset when both First_Email and Summary are used. Ultimately, it was determined that the best performance was achieved when fields were assigned an equal Field Boost Factor towards the similarity ranking between defects.

TABLE II
AVERAGE FIELDS LENGTH

Field	Average Field Length (Term)
Summary	8
Description	15
First_Email	103

C. Search Parameters

For further evaluation and parameter modification, the best performing fields will be used: First_Email and Summary. Two very common techniques to reduce noise and generate higher recall rates are removing stopwords and including synonyms. Using a customized list of Stopwords and Synonyms, evaluations were performed to determine the effectiveness of these techniques. As can be seen in Figure 4, there is virtually no difference in recall rate when removing Stopwords. The use of a synonyms list decreases the recall rate by introducing noise in the results.

Due to the difficulty in creating a list of Stopwords large enough to encompass all of the words/terms that would generate noise, we employed a new parameter to be used only on the searching side (no indexing changes needed). This parameter is associated with the MoreLikeThis Handler, and is called Max. Doc. Freq. When set to X%, any words that appear in greater than X% of all documents will be ignored. It can be thought of as a pseudo-Stopwords implementation that dynamically sets Stopwords based on the dataset you are using. As shown in Figure 5, this parameter was tuned until the optimal value of 5% was determined with the highest recall.

Through a similar procedure, we determined the optimal parameter values for the MoreLikeThis handler which is used in the Searcher. The minimum frequency of a term for it to be considered within a select document was found to be optimal at 1. Likewise, the minimum frequency of documents that a word must appear in for it to be considered was also found to be optimal at 1. The minimum word length was found to be best when it was set to 2 characters.

D. Similarity Search Algorithms

After optimizing data fields and search parameters, the question of whether the default similarity search algorithm (i.e., TF-IDF) is optimal was put to test. Two Language Model ranking methods - Jelinek-Mercer smoothing and Dirichlet smoothing were tested alongside the Okapi BM25 ranking

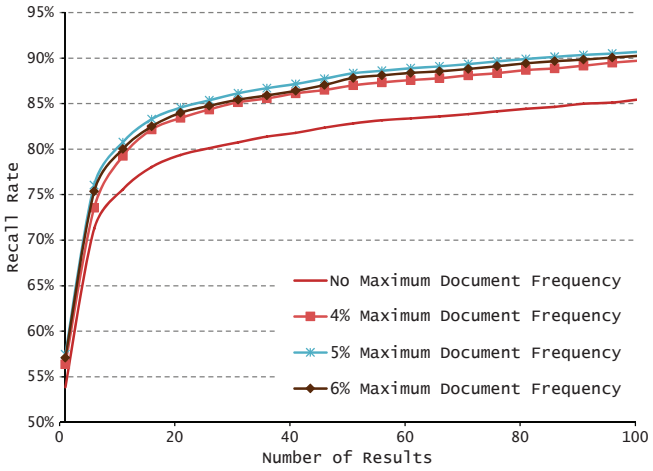

Fig. 5. Recall Rates with Dynamic Stopwords

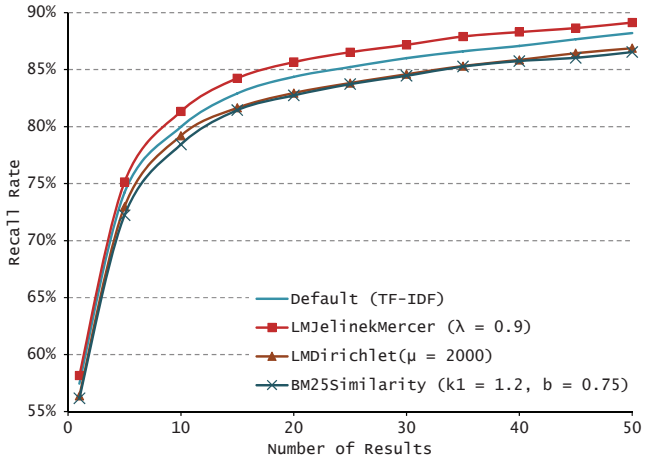

Fig. 6. Recall Rates with Alternate IR Algorithms

model. To use these algorithms, we must specify their input parameters: λ for LMJelinekMercer, k1 and b for BM25, and μ for Dirichlet. Figure 6 compares the recall rates of alternate search algorithms on our dataset against the default TF-IDF employed by Lucene. We note a globally higher recall rate when using the Jelinek Mercer language model.

It has been determined that the optimal value of λ for long fields is 0.7, and for short fields it is 0.1 [19]. Also, referring to the field lengths recorded in Table II, we note that the field length for First_Email is very long with 103 terms, but only 8 terms for Summary. Considering the complete defect report as a single field, we find that the best recall rate is actually achieved when setting λ to 0.9.

Since we notice an improvement in the recall rate when employing alternate search algorithms, we also test these parameters on another BlackBerry defect repository (data and results are not presented in this article) in an effort to try and improve the recall rate for other kinds of defects. It is noteworthy, that for this alternate defect repository, we found the best results is achieved with $\lambda = 0.5$. This is

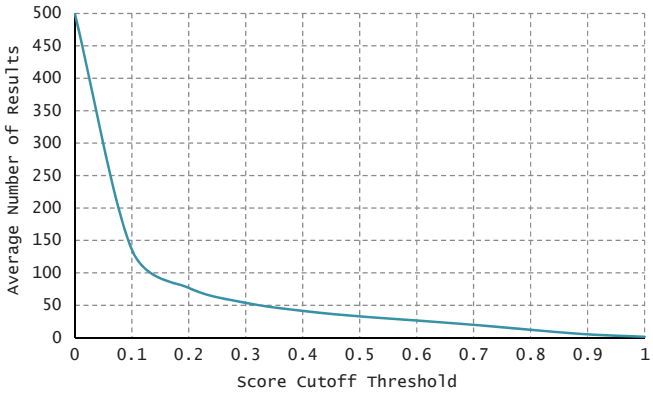

Fig. 7. Precision of Results with Score Cutoff Thresholds

consistent with the relative difference in the field lengths that we calculated between the two datasets. The previously determined value of 5% for the Max. Doc. Freq. is still found to be optimal when considering these alternate algorithms (and the results reflect the use of this parameter value).

E. Setting Cutoff Threshold

Generally, because Lucene does not have a normalized method of computing scores, limiting the search results based on the score threshold is not a recommended practice for a couple of reasons (e.g., as argued in [20]). Most importantly, the result set evaluated against by using a certain score threshold would likely undergo a substantial change after several reindexes of the data. Therefore, to be able to use a limiting score threshold, we must first determine if changes in the index would affect the trend of our recall rate evaluations against each score limit. In other words, by testing a current index against an index from some sufficiently set time in the past, we can determine the viability of using a limiting score threshold. We can only proceed to use a method of limiting scores if both evaluation results yield the same shape for the recall plots (which was the case in our analysis). Figure 7 shows the effect of applying score threshold on the average number of the returned results.

Figure 8 shows the recall rates and precision when we vary the minimum score threshold of search results and also apply a maximum number of results to display. The secondary cutoff point is varied with a maximum of 10 and 50 search results, to determine which is the better option. Here, 10 is the recommended size of the resultset by other researchers such as [5], and 50 is the maximum desired resultset size specified as a part of DDD's requirements.

As we can see in Figure 8, there is a tradeoff between recall and precision. Hence, in an attempt to determine the best threshold setting we used F-measure. F-measure can be interpreted as a weighted average of the precision and recall, which reaches its best value at 1 and worst score at 0. Here, we use the general formula of F-measure for positive β [12]:

$$F_\beta = (1 + \beta^2).\frac{precision.recall}{(\beta^2.precision) + recall} \quad (1)$$

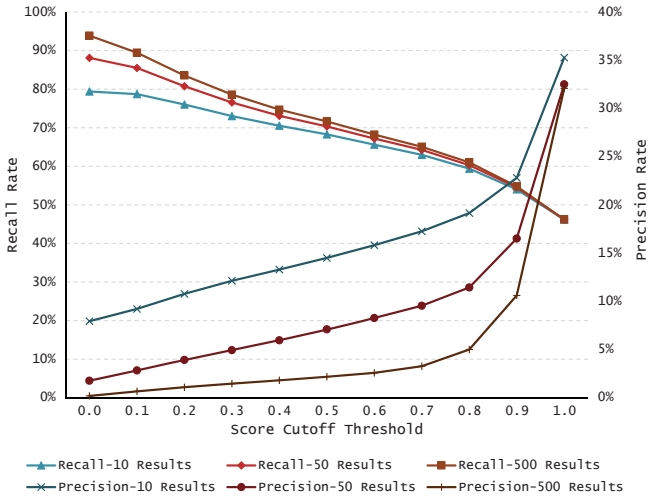

Fig. 8. Recall and Precision Rates with Score Cutoff Thresholds

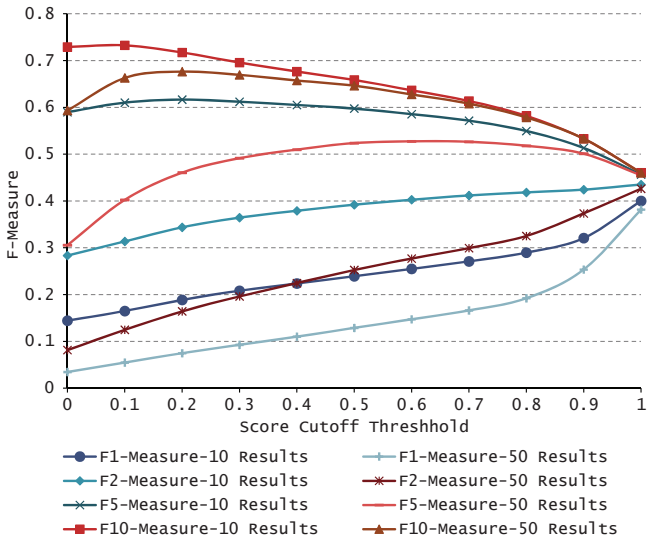

Fig. 9. F_β Measure with Score Cutoff Thresholds and Fixed Results

The correct β has to be set for an effective measure. In the case of duplicate defect detection, we are aware that more emphasis needs to be put on recall than precision. However, the exact favorable value for β has to be specified for each project and dataset. Thus, we set out to test greater levels of F-Measure when recall is weighted higher than precision. Figure 9 displays F-measures where recall importance is varied from 1-5 times higher than precision (i.e., $\beta = [1, 2, 5, 10]$) for fixed cutoff threshold of 10 and 50.

For selecting a desirable score threshold, the desirable trend needs to be selected. This decision is dependent on two requirements: i) the maximum desired size of the result set (e.g., 10 for a single result page), and ii) the importance of showing all duplicate pairs even if we have to show many more results. Then, the parameter tuner can pick the score threshold value which maximizes the selected F-measure.

TABLE III
NUMBER OF TRUE DUPLICATE DEFECT PAIRS PER TIME FRAME

Time Frame	Percentage of True Duplicate Pairs
90 Days	99.39%
77 Days	99.27%
30 Days	95.39%
15 Days	89.18%
5 Days	81.06%
1 Day	69.53%

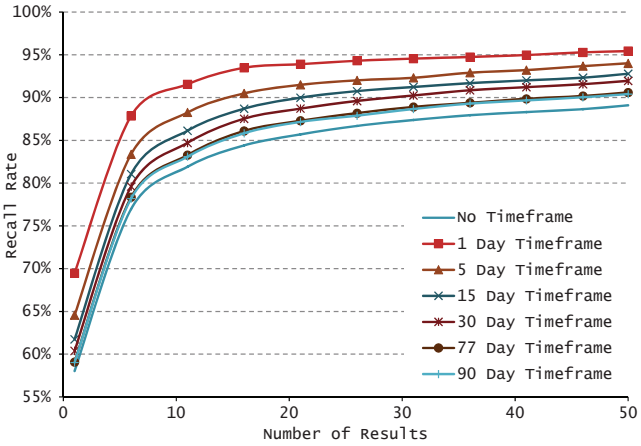

Fig. 10. Recall Rates with Timeframes

F. Further Inspection of Defect Reports

Along with the stored fields that we use to analyze for similarity, there are also stored fields that are not analyzed. Besides the ID, a notable field is the creation date of the defect. Indexing the creation date for each defect allows us to set another parameter that will serve to analyze the timing of duplicate defect reports, as well as potentially increase recall and precision. We can restrict the list of true duplicate defect pairs to those that are within a certain time frame of each other, and also restrict the search results to be within this time frame as well. The question that this serves to answer is whether or not the duplicate defects are condensed into a certain time frame from their respective original defects, and whether that time frame is sufficient in that it will encompass the vast majority of true duplicate defect pairs. As shown in Figure 10, the recall rate increases as the time frame is lowered. Note that for these time frame evaluations, we employed the best searching algorithm (i.e., Jelinek-Mercer smoothing).

Since the time frame parameter also filters true duplicate defect pairs beyond the time frame, in order for this parameter to be useful, we need to choose a time frame that includes a sufficient percentage of true duplicate defect pairs. For this reason, we analyzed the average distribution of time differences between each duplicate pairs. Average time distances for our data set had a normal distribution as shown in Table III.

The percentage values in Table III suggests what percentage of duplicate pairs will be filtered out by setting different timeframe values. In our case, we decided to set it to 77 days

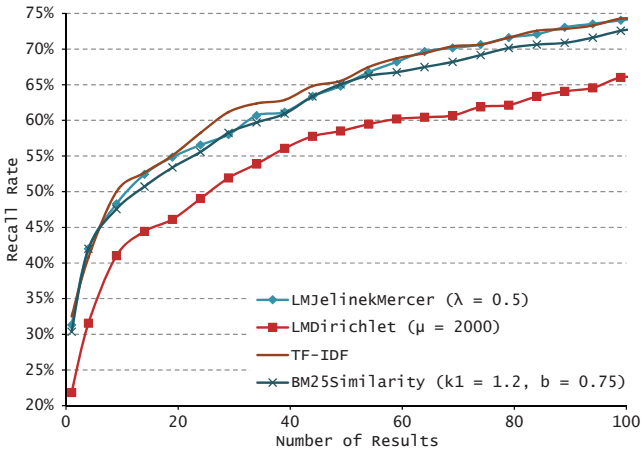

Fig. 11. Firefox Recall Rates with Alternate IR Algorithms

timeframe based on user requirements. This decision comes at the cost of filtering out 50 true duplicate defect pairs, while we observe a 1% increase in recall rate, as shown in Table III.

G. The Mozilla Dataset

After tuning DDD parameters for the BlackBerry dataset, we applied our approach on the Mozilla defect repository. There are two reasons to perform this analysis. First, it will examine the reusability of our approach, and help us gauge the effort required to use this approach on a project we have limited knowledge of. Secondly, we can study if there exists a fixed optimal value for each tuning parameter, or whether tuning and customization of duplicate defect detectors is needed for each project.

Starting with the optimal parameter set for BlackBerry repository, we tried to improve the performance of DDD for the Mozilla dataset. Figure 11 shows recall rate when using alternate ranking algorithms on an index of Mozilla defects, taken from the time frame June 2007 - December 2007. The results demonstrate worse performance compared to the BlackBerry dataset. We observe that LMJelinekMercer is still the best algorithm, however with a different $\lambda = 0.5$. Also, it is interesting to see that the default TF-IDF, which has no language model, has quite a similar performance to LMJelinekMercer. Additionally, the optimal maximum document frequency that is employed with all of the algorithms for detection of Mozilla Firefox duplicate defects is 20%.

IV. Discussion on Obtained Results

The experimental studies allow us to analyze the impact of each tuning parameter in duplicate-defect detection performance for the selected datasets. In this study, we identify a set of parameters that are usually left out in search-based software engineering approaches, while their default values are not necessarily optimal for the input dataset. We also show that some parameters can greatly impact the search quality, but their optimal value may not be the same for all datasets and their impact is not constant. For example, among all tuning parameters, a particularly notable parameter was the Maximum Document Frequency. It's optimal value for BlackBerry dataset is 5% while for Mozilla Firefox is 20%. By examining the obtained results and reflecting upon the lessons learned from the experimental studies, our findings can be summarized as following:

a) Global vs. Case-dependant Optimal Parameter Values: By observing the properties of defect repositories, we could raise the hypothesis that for some parameters there exists a global optimal value that is independent from its input dataset. However, at this point, we cannot support this hypothesis and list the parameters for which there is a fixed optimal value regardless of the input dataset and user requirements. However, we believe experiments can be designed to support and prove this hypothesis.

b) Varying Search Quality: We believe this high fluctuation in performance from case to case is mainly due to the nature of the project and the defect reporting practices. The major part of Mozilla is Firefox web browser and its related technologies. On the other hand, the selected BlackBerry defect repository contains a much wider range of applications and features. Although we haven't performed statistical analysis at a defect-report level, comparing the nature of these two large defect repositories suggests that search-based duplicate detection performs better when defect reports are less similar in nature and covers a more diverse feature set. It also reveals the importance of accurate and detailed defect reporting in large and long-lived software projects.

c) Search-time vs. Index-time Tuning: The bulk of parameter tuning for our approach is done at search-time rather than index-time. The task of varying the fields used in the similarity search is most efficiently performed at search-time. However, when the data fields to be used are finalized, we can re-index the data to only include the required data fields with appropriate index-time boost factors. This action will reduce the size of the index, and can possibly reduce search time. As the goal of this study was to improve the search quality and the search time in all scenarios was within an acceptable range ($< 500ms$), we did not study this potential improvement. However, search response time can become a bottle neck in project peak times (e.g., before and after release) in large software organizations. Therefore, reducing the response time is another interesting dimension to further improve search-based duplicate defect detectors.

d) Recall vs. Precision: Measuring recall shows how many duplicates the search component is able to find 'in total' out of all the duplicates for a given defect ID. However, recall falls short when we have more than one duplicate pair per defect and we are interested to select and report top-N results to improve precision. The reason for this is that defects with more duplicate pairs could be measured with lower recall than their actual recall value. For example, suppose we have three different defects (b1, b2, b3) with different number of duplicates: 1, 10, and 100 respectively. If we flawlessly top rank duplicates and return top-10 results: the recall for b1, b2, and b3 will be:

- b1: 1st row is the duplicate, and 9 non-duplicates, recall rate is 100%.
- b2: 10 out of 10 are duplicates, recall rate is 100%.
- b3: 10 out of 100 duplicates, recall rate is 10%.

If we consider the complete resultset for each defect, we will have the actual recall rate of 100% for all three defects. To address this discrepancy, we first tune the search component (to improve recall) and then we tune the selection component (to improve precision) while maintaining the recall. Additionally, for b1, b2, and b3 the precision will be:

- b1 is 10%, as you have returned 9 false results.
- b2 is 100%, 10 out of 10 correct.
- b3 is 100%, 10 out of 10 correct (although we have more to report, the rest are out of bound).

This example shows that a fixed recall rate (not increasing) within a specific range for result set size, means a loss in precision. Precision can fluctuate with an increase in the size of result set. This means that for each defect there exist one or more 'number of returned results' (i.e., cutoff threshold) that maximizes the precision for that input defect. This analogy highlights the importance of finding optimal cutoff threshold that can maintain recall and improve the precision. To overcome the mentioned above shortcomings of recall and precision, other metrics such as F-measure and Matthews Correlation Coefficient (MCC) can be used instead [21].

V. Related Work

Broadly speaking, the two popular research approaches so far to assist with duplicate defect report detection efforts have evolved around: i) providing a list of top-N similar defect reports which a triager would then assess to find the true duplicates, and ii) preventing duplicates from reaching developers through automatically filtering. In the latter approach, there is no triager involvement and it is assumed that the filtering process is accurate.

Runeson et al. [5] used the Vector Space Model (VSM) to detect duplicate defect reports. They experimented with this technique on defect reports from Sony Ericsson. In the best case, they achieve a recall rate of 39% for a list of size 10 and 42% for a list of size 15. They were able to achieve a maximum of 60% recall rate using a very large list size.

Wang et al. [1] manually create execution traces from defect reports by reproducing defects and apply the VSM to detect duplicates. On their data set, they were able to achieve a recall rate of 67-93%. The drawback to their approach is the additional overhead of reproducing the defects to get the execution traces, which also has storage implications on the defect repository.

DebugAdvisor [8], developed by Microsoft researchers, is a search-based tool to find related bugs and other related information such as people and files. Similar to DDD, DebugAdvisor allows users to search based on complete defect information (i.e, fat query), which contains all contextual information for a given defect. In addition to recall and precision they also presented qualitative and quantitative feedback from over 100 users inside Microsoft.

Sun et al. used discriminative models to detect duplicate defect reports [7]. By treating defect report duplication as a binary classification problem, they classify defects into two sets- duplicates and non-duplicates. Furthermore, they apply the SVM classifier on the defect reports to determine the likelihood of a report to be a duplicate and organize the reports into "buckets". An incoming defect report is added to its corresponding bucket if its a duplicate, or a new bucket is created for the defect report if it is not a duplicate. Their techniques outperformed the state of art results by 17-31% on their Open Office data set, 22-26% on their Firefox dataset and 35-43% on their Eclipse data set.

Kaushik et al. [9] study the comparative performance of Information Retrieval models, specifically word based models such as the VSM and its variants and topic based models such as LDA and LSI. Their results show a recall rate of 60% for the Eclipse and 58% for the Firefox datasets for top-10 results. They find that word-based models, particularly a Log-entropy based weighting scheme, outperforms topic based models.

Nguyen et al. [22] leverage IR-based features and topic-based features by modeling a defect report as a textual document describing certain technical issues. Using historical data, their approach learns the sets of different terms describing similar technical issues. The evaluation shows an improvement over the state-of-the-art approaches by up to 20%.

Sun et al. [15] propose a retrieval function and extend the BM25F algorithm. They optimize the retrieval function for specific defect repositories through supervised learning. They validate their approach on the Mozilla, Eclipse and OpenOffice projects and showed a relative improvement of 10-27% in recall rate and a 17-23% improvement in precision.

Jalbert et al. use a graph clustering algorithm and text similarity to identify and filter duplicates [6]. They employ a linear regression model to make a binary classifier to distinguish duplicates from non-duplicates. They were able to detect 8% of the duplicates. Their study indicates that semantically rich textual information contributed more than surface features in helping detect duplicates. Of these, the title (summary field) and the defect report description were the most important distinguishing features. In a recent work, Tian et al. extend the original approach by improving the accuracy of automated duplicate defect report identification [23].

VI. Conclusion and Future Work

In this article, we presented the designed and developed Duplicate Defect Detection (DDD) framework for BlackBerry. DDD in its current state is operational and is integrated with BlackBerrys defect tracking system. Through a custom button, any user in the defect tracking system can search for potential duplicates on a new defect or an existing defect.

The proposed approach is generic and incorporates a feedback loop for parameter tuning and customizing the search engine for individual datasets. As assessing all the possible combinations of tuning parameters can be exhaustive, we optimize the recall with the help of heuristics (i.e., line search algorithm). For our data set, we found a set of parameter values

and algorithms that improve the results, while their values need to be tuned for each project. Our experimental results highlight the high impact of parameter tuning on the system's performance. However, the employed one-dimensional optimization technique for finding optimal parameter values can be extended to more advanced heuristics (e.g., Genetic Algorithms) that can explore the search space more efficiently, and suggest pareto optimal values when optimizing for conflicting objectives (i.e., measures).

Another factor that we tried to address in this study is to finely tune the cutoff value for the most favorable number of the defects to be reported as duplicates. In this work, we performed F-measure analysis to find optimal cutoff points based on the average recall and precision for all input defects. However, this can be improved by tuning the cutoff threshold for each individual defect and its result set.

For the performance evaluation, we mainly calculate the trend of recall for a range of returned results and report the average over all defects. The main rationale behind this decision were the requirements from BlackBerry that state maximum recall as the top objective. However, in our studies we did not consider the order in which the returned defects are presented. Having the ground truth, evaluation could be improved by applying other measures that regard position in the ranked sequence of documents such as Mean reciprocal rank (MRR), Mean Average Precision (MAP), and Discounted Cumulative Gain (DCG) [12].

Moreover, by tuning DDD for various types of projects and their defect repositories we will gain insight of the how project and defect characteristics impact tuning parameters. As a future work, predictive modeling techniques, as described in [24], can be used to elicit this knowledge and help us to refine it in a form of patterns. For example, there could be some duplicates that can never be detected for all possible values of a parameter, while there are also some duplicates that will be detected for some parameters values. Studying the characteristics of these two different kinds of duplicates is an interesting direction to further improve this research.

ACKNOWLEDGMENT

We thank Mark Moore and Blair Cooper at BlackBerry for providing many invaluable feedbacks, guidance and encouragement, Shawn Burke for integrating the DDD framework with existing defect tracking system at BlackBerry. We are also grateful to Rebecca Cummings, James Carney and his team at BlackBerry for participating in the trial phase of DDD and providing improvement suggestions.

REFERENCES

[1] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2008, pp. 461–470.

[2] Y. C. Cavalcanti, E. S. Almeida, C. E. Cunha, D. Lucredio, and S. Meira, "An initial study on the bug report duplication problem," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2010, pp. 264–267.

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

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

[5] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2007, pp. 499–510.

[6] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems." in *Proceedings of Dependable Systems and Networks (DSN)*, 2008, pp. 52–61.

[7] C. Sun, D. Lo, X. Wang, J. Jiang, and S. C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2010, pp. 45–54.

[8] B. Ashok, J. Joy, H. Liang, S. Rajamani, G. Srinivasa, and V. Vangala, "Debugadvisor: a recommender system for debugging," in *Proceedings of European Software Engineering Conference and Symposium on The Foundations of Software Engineering*, 2009, pp. 373–382.

[9] N. Kaushik and L. Tahvildari, "A comparative study of the performance of ir models on duplicate bug detection," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2012, pp. 159 –168.

[10] M. Harman, P. McMinn, J. Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical Software Engineering and Verification*, ser. Lecture Notes in Computer Science, B. Meyer and M. Nordio, Eds., 2012, vol. 7007, pp. 1–59.

[11] O. Gospodnetic, E. Hatcher, and M. McCandless, *Lucene in Action*, 2nd ed. Manning Publications Co., 2010.

[12] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press Cambridge, 2008, vol. 1.

[13] J. Russell and R. Cohn, *Precision and Recall*, 2012. [Online]. Available: http://books.google.ca/books?id=NLOYMQEACAAJ

[14] A. Eiben, Z. Michalewicz, M. Schoenauer, and J. Smith, *Parameter Setting in Evolutionary Algorithms*, ser. Studies in Computational Intelligence. Springer Berlin Heidelberg, 2007, vol. 54, ch. Parameter Control in Evolutionary Algorithms, pp. 19–46.

[15] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proceedings of International Conference on Automated Software Engineering*, 2011, pp. 253–262.

[16] A. E. Eiben and S. K. Smit, *Autonomous Search*. Springer, 2012, ch. Evolutionary Algorithm Parameters and Methods to Tune Them, pp. 15–36.

[17] D. Luenberger and Y. Ye, *Linear and nonlinear programming*. Springer, 2008, vol. 116.

[18] J. Parapar, M. Vidal, and J. Santos, "Finding the best parameter setting: Particle swarm optimisation," in *Proceedings of Spanish Conference on Information Retrieval*, 2012, pp. 49–60.

[19] C. Zhai and J. Lafferty, "A study of smoothing methods for language models applied to ad hoc information retrieval," in *Proceedings of International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2001, pp. 334–342.

[20] Lucene-Java Wiki, "Scores as percentages," http://wiki.apache.org/lucene-java/ScoresAsPercentages, 2009.

[21] M. Shepperd and S. MacDonell, "Evaluating prediction systems in software project estimation," *Journal of Information & Software Technology*, vol. 54, no. 8, pp. 820–827, 2012.

[22] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Proceedings of International Conference on Automated Software Engineering*, 2012, pp. 70–79.

[23] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," in *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*, 2012, pp. 385–390.

[24] M. Harman, "The relationship between search based software engineering and predictive modeling," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, 2010, pp. 1–13.