Is Text Search an Effective Approach for Fault Localization: A Practitioners Perspective

Vibha Singhal Sinha, Senthil Mani and Debdoot Mukherjee
IBM Research – New Delhi, India
{vibha.sinha, sentmani, debdomuk}@in.ibm.com

Abstract

There has been widespread interest in both academia and industry around techniques to help in fault localization. Much of this work leverages static or dynamic code analysis and hence is constrained by the programming language used or presence of test cases. In order to provide more generically applicable techniques, recent work has focused on devising text search based approaches that recommend source files which a developer can modify to fix a bug. Text search may be used for fault localization in either of the following ways. We can search a repository of past bugs with the bug description to find similar bugs and recommend the source files that were modified to fix those bugs. Alternately, we can directly search the code repository to find source files that share words with the bug report text. Few interesting questions come to mind when we consider applying these text-based search techniques in real projects. For example, would searching on past fixed bugs yield better results than searching on code? What is the accuracy one can expect? Would giving preference to code words in the bug report better the search results? In this paper, we apply variants of text-search on four open source projects and compare the impact of different design considerations on search efficacy.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: [Debugging aids]

General Terms Experimentation,Measurement

Keywords Empirical Study, Bug-Solving

1. Introduction

Identifying buggy code fragments can be particularly time-consuming and tedious; statistics suggest that over half of the total time in any software project is spent in locating and fixing bugs [16, 19]. To address this problem, many automated fault-localization techniques based on static and dynamic program analyses have been developed. The most widely researched approach in the area is based on program slicing (e.g., [3], [4], [10], [14]); other approaches include statistical debugging (e.g., [7, 13, 15]) and delta debugging [23, 24].

A different class of debugging techniques, based on text analysis, uses the text in the bug report to recommend source files that potentially need to be fixed. Given the bug report for a new bug, one may search the project's bug repository for similar bugs resolved in the past to get a better understanding of the problem and a potential resolution of the bug-at-hand (e.g., [5, 8]). If source files were fixed to resolve similar bugs, then these may be recommended for the new bug as well. Yet another way to apply text-search is to directly search on the code repository with the text of the bug report taken as a query (e.g., [1, 11, 20]). The underlying principle here is that the bug report often contains references to code terms—class names, function names, variables etc. Also, comments in the code are written as free flowing text, so it is likely that we may find common words between bug reports and code comments.

The text-search based approaches can be applied more generically than the rigorous program analysis techniques since they are not restricted by the programming language used in the application or limited by the presence of test cases to reproduce the fault. However, they may suffer in case: (1) there is a low overlap between the vocabulary of the bug report used to query and that of the search repository. Enslen et. al. [2] suggest a way to increase vocabulary overlap between bug reports and code through *identifier splitting*, whereby all code terms are converted to their componentized words. For example, the code word *TextfieldTool* would be translated to three words: *text, field, tool*. (2) the bug repository does not record linkages to code fixes necessary to resolve bugs. However, increasingly project teams are realizing the benefits of preserving this linkage and adopting bug management systems such as Jira or Rational Team Concert, which allow linkage with version management repository. In these tools, the developer can link a bug with the code change set that was committed in the version management system to fix the bug.

Most of the existing work in applying text-search based techniques has been evaluated on small subjects only. In *DebugAdvisor* [5], the authors use a proprietary Microsoft application as their subject and search on a repository of past bugs. They evaluated the precision and recall of search for 50 bug queries. Rao and Kak [20] used the iBUGs dataset (291 bugs), which primarily has bugs from AspectJ and Rhino. They search on source code, which is pre-processed using identifier splitting. Others [1, 18] have reported results of searching on the code repositories of Mozilla, Eclipse, Rhino and JEdit. However, the number of bugs used in these evaluations are very low—5 to 15 bugs. Prior art [1, 11, 20] studies the search performance of different language models (e.g., Unigram, Vector Space Model, Latent Semantic Indexing) in great depth. In fact, Rao and Kak [20] empirically study (using a test suite of 291 bugs) that simple language models such as VSM provide equivalent to the more complex models such as LDA.

One of the biggest drawbacks of the existing works is that because of the small test suite size, it is difficult to make a judgment on general applicability of the technique. For example, in [1], the authors reported when they searched for 3 Eclipse bugs on the code repository they were able to find a correct file recommendation in top 2 search results. They used the LDA language model for search and claimed that it worked better than LSI, which returned the correct match for each of these three bugs in top 7 search results. However, when we applied VSM model on 815 bugs from a sub-project in Eclipse, we found a correct file returned only for 10% (80) bugs in the top 7 results. Considering [20] already showed that VSM provides equivalent efficacy to LDA, this makes us believe that the 3 eclipse bugs evaluated in [1] were not indicative of the general population of bugs. The aim of this paper, is to provide practitioners a truer picture of both lower and upper bound efficacies of text search, by doing an empirical study on large number of bugs from multiple subjects.

Another gap in the existing published literature is that none of the efforts perform a comparative study on the two variants of the search approach–*searching on past bugs* and *searching on code*. Also, no prior work discusses the effect of pre-processing techniques (e.g., identifier splitting) or bug characteristics (e.g. size of bug, number of files modified to fix the bug etc) on the effectiveness of the text search recommendations.

In this paper, we address these issues through an extensive empirical study of four large open-source projects - *BIRT, Eclipse-Datatools, Hadoop and Derby*. Overall, we have a test data-set comprising of 1177 bugs. Our goal is to establish statistically grounded empirical evidence about the impact of different design criteria (for the search approach) on the productiveness of search-based recommendations for bug solving. We believe that such a study is necessary to make the idea of using text-search for bug solving more widely acceptable. It also helps us better understand the available design options in terms of their benefits and tradeoffs. Our empirical study aims to study the following.

- We compare the efficacy of three techniques, each of which search over a different kind of search repository - (1) the collection of all bugs resolved in the past, (2) the code base, and (3) a version of the code base processed through identifier splitting. In all the three cases, the complete text of the bug report, comprising of the title and the description, is taken as a query. We determine whether the accuracies of these techniques are better than chance and whether they are similar or complementary in terms of their recommendations. We find that there is no clear winner between *searching on code* and *searching on past bugs*. When the search result set size is taken to be 5, the precision varies from 4% to 13% across our four subjects while searching over bug repository versus 4% to 5% while searching over code. For comparing different techniques, we introduce a new metric called *Bug Coverage* defined as the percentage of bugs where the approach returns at least one correct file match for a given search result size. The bug coverage varies from 30% to 54% across our subjects.
- We analyze the possibility of combining the results of the three techniques to make more effective recommendations, more consistently. Athough, the efficacy of an average search over past bug repositories is similar to that over code repositories, we find that the techniques are complementary because one technique may score over another for certain bugs, and the other may prove to be superior in certain other cases. We experiment with different ways to combine the search results from these techniques in order to form a better final result set (improvement in bug coverage varying between 1% and 46%).
- We study the impact of various query construction strategies (boost to title words, code terms etc.) on search performance. We find that giving a preference to words in the title improves the bug coverage from 1.6% to 7.8% across our subjects for a search result size of 5. Further, we note the correlation between coverage and other physical features of a bug report such as total number of words in a bug report, number of code words and title words.

The main contributions of the paper are: (1) An empirical evaluation of text-search based fault localization on four open source projects. We compare various search design criteria of index creation and query construction. (2) A novel way of combining *search-over-bug-history* and *search-over-code-base*, that achieves greater bug coverage than the individual techniques themselves. (3) An evaluation of whether different bug features correlate with search efficacy.

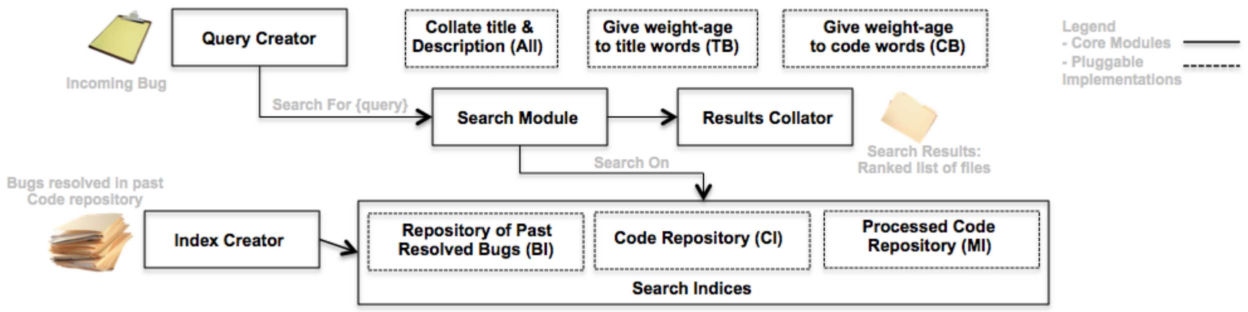
Query Creator
Incoming Bug
Collate title & Description (All)
Give weight-age to title words (TB)
Give weight-age to code words (CB)
Legend
- Core Modules
- Pluggable Implementations
Search For {query}
Search Module
Results Collator
Search Results: Ranked list of files
Search On
Bugs resolved in past
Code repository
Index Creator
Repository of Past Resolved Bugs (BI)
Code Repository (CI)
Processed Code Repository (MI)
Search Indices

Figure 1. Search Approach. The framework contains four core modules: (1) Query Creator, (2) Index Creator, (3) Search Module, (4) Results Collator. The implementations of these modules is pluggable. A module can have more than one implementation depending on the choice of design considerations.

Rest of the paper is organized as follows. In the next section, we outline the search framework we developed for our experimentation. In Section 3, we present the results of empirical evaluation of the various search considerations on four open source projects. Section 4 outlines the related work in the area of debugging and fault localization. Finally, in Section 5 we summarize our findings.

2. Approach

Figure 1 outlines the search framework implemented for our evaluation of the different text-search methods that recommend relevant source files for an incoming bug. It is composed of: (1) an *index creator* with a pluggable search repository, which processes *documents* to create *indices* [17] that make searching easier[1]; (2) a *query creator*, which processes the text in the incoming bug report to form a query with pluggable design variant; (3) a *search module*, which fetches documents from the indices that are similar to the query. When searching over the code base, the source files returned in the set of search results can be the recommendations from our system. However, a search over the corpus of bug reports yields a set of bug reports and the recommendations are the source files associated with these bug reports. (4) a *result collator*, which combines the results returned from the different repositories to increase the relevance of recommendations. We choose the Vector Space Model (VSM) to design our search system. Rao and Kak [20] have shown that VSM works no worse than other models of information retrieval (e.g., Latent Semantic Indexing, Latent Dirichlet Allocation) that have greater sophistication.

Index Creation: We experiment with three kinds of indices: (1) *Bug Index (BI)* indexes terms extracted from past bug reports; (2) *Code Index (CI)* indexes the source code files (taken as-is); and (3) *Meta Index (MI)* indexes a processed version of the source files–making them closer to natural language text. In *MI*, we store documents created from the *code terms* and comments present in each source file. The *code terms* include names of packages, classes, class variables, methods, formal arguments and method variables. Further, *identifier splitting* [20] is applied to each code term in order to convert the *term* to its constituent words. For example, the code term *getName* is translated to *get* and *name*, and the code term *com.xxx.foo.TestClass* is translated to the words *com, xxx, foo, test* and *class*. The splitting is done by separating words based on camel case rules. Note that code terms might also occur as part of comment text and these are not split. The extracted code terms are also stored separately as the *code dictionary*, which is later used during the query creation phase.

Query Creation: We create a query vector from the terms present in the title and description of the bug report. As a pre-processing step, we remove the stops words such as *to, is, and* etc and *Java* keywords such as *java, package, class* etc. Then, we compute TF-IDF scores for all query terms and create the query-vector. This is our first querying strategy - *All (A)*. For fine tuning search performance, we employ two more querying strategies, *Code Boost (CB)* and *Title Boost (TB)*. In the *CB* querying technique, each term in the query is matched against the code dictionary. The weight of matching terms is given a boost relative to others. Similarly, in *TB*, we update the query vectors by increasing the weight of terms that come from the bug title.

Search : We use Apache Lucene's implementation of a VSM based full text search engine to host our index creation and query creation strategies. We search each of our indexed repositories *CI*, *MI* and *BI* separately by applying the querying strategies *A*, *CB* and *TB*. For a given query vector, Lucene returns a similarity score for every document in the chosen index. The scores indicate a relevance rank of the document with respect to the query. For each query that is run over *CI* and *MI*, the top *X* search results are returned as recommendations. However in case of *BI*, the search results

[1] A *document* is any text file in the repository being searched. In our case, the repository may be a code base (either as-is or processed) or a collection of past bugs. An index is a mapping of words (a.k.a terms) to the locations in the documents where they are present.

Subjects	Releases	Total # of Bugs	# of Bugs in test-set	# of Files in release	# of Bugs in repository
	2.5.0			6351	
Birt	2.5.1	1777	815	6524	
	2.5.2			6633	21064
	1.0			1925	
Datatools	1.5	1130	93	2379	
	1.6			2968	2698
Derby	10.5.3.0			1687	
	10.8.1.2	242	136	1742	3492
Hadoop	0.20.0			837	
	0.21.0	191	133	1328	3879

Table 1. Details of the subjects used for our experiments

are bug reports; so we return all the source files fixed for the top *X* similar bugs as our recommendations[2]

Result Collation : This module combines the results returned from searching different repositories. The objective is to combine the results optimally to increase the number of bugs for which we return at least one correct recommendation without increasing the size of the recommendation set. We present different heuristics to combine the recommendations in Section 3.3.2.

3. Experiments

We evaluate the different search indices and querying strategies described in Section 2 on four open source projects. In this section, we describe the experimental subjects and the method used in our study. Next, we present the empirical results and analyze them in order to answer the following research questions:

- **RQ1 (Effectiveness)**: How does searching on —Bug Index (BI), Code Index (CI) and Meta Index (MI), fare in terms of the effectiveness of the recommendations produced by them? Are they just as good as chance or any better? Also, how do they compare with one another?
- **RQ2 (Combination)**: How can we combine the sets of recommendations from the three search indexes to increase the bug coverage without sacrificing the accuracy of the resultant recommendation set (as measured by F1-score)?
- **RQ3 (Feature Impact)**: How do different aspects of the source code and the bugs available in a project impact the effectiveness of search?

3.1 Experimental Data & Setup

We select our experimental data-set from four open source projects, *Birt* [3] and *Datatools*[4] of Eclipse and *Derby*[5] and *Hadoop*[6] of Apache. For each of these projects, we obtain the list of fixed bugs from its bug management system and code for those releases that record a high number of the bugs relative to the project.

To create the experiment oracle (or ground truth), we need to know the source files that were modified to fix the bugs. The Apache based projects use a bug management system called *JIRA*[7], which keeps a record of the files fixed to resolve a bug. For the Eclipse projects, we trace the linkages from bugs to the buggy source files by mining their version management systems. We study the version logs to define regex patterns that are able to detect bug identifiers mentioned as part of comments inserted during code commits (an approach followed in [8]); e.g, if the commit-comment contains “#234561” or “Bug-234561”, then it indicates that the code change is related to the bug – 234561.

For each of the four subjects, we prepare a test-set such that it consists of only those bugs that are reported against one of our chosen releases and has at least one source file associated with them. Table 3 gives details for each subject—the selected code releases, the total number of bugs reported for those releases, the number of bugs selected from total numbers of bugs available to create the test set, the total number of *java* files available per release and the total number of fixed bugs in the project’s bug repository. The total number of bugs in our test-set for our experiments across all four subjects is 1177 bugs. This is 35% of the total bugs available for the subjects across the 10 releases that we consider (column (2) of Table 3). We ignore the remaining 65% as either they were not associated with any changes to source (.java) files (for JIRA based bugs) or we were not able to infer the association (for Eclipse projects bugs). However, 1177 bugs can be considered to be a significantly large test data set in comparison to the same for prior work [1, 5, 18, 20]. For each subject, the size of the test-data is 3-4% of its corresponding bug repository.

Further, we index all bugs in the subject’s bug repository to create three different search indices –*BI*, *CI* and *MI* (as explained in Section 2). For each bug in the test-set for the subject, we create three different queries following the strategies—*A*, *CB* and *TB*.

Depending on the research question to be answered, we execute different *search techniques*—combinations of an indexing technique and a querying strategy. For example, the search technique $\{MI{:}TB\}$ means applying the querying strategy TB to search the index MI. The source files returned in the recommendation sets produced by a search technique are matched against the actual fixed files (ground truth) in order to compute the following measures:

- *Bug Coverage:* The percentage of bugs in the test set for which the search returns at least one file in the recom-

[2] If the search results return a bug, which in turn has greater than 10 files modified, we do not include any files from that bug in our search results. This was done to remove any un-toward positive bias in calculating efficacy of BI.

[3] `http://www.eclipse.org/birt/phoenix/`

[4] `http://www.eclipse.org/datatools/`

[5] `http://db.apache.org/derby/`

[6] `http://hadoop.apache.org/`

[7] `http://www.atlassian.com/software/jira/`

mendation (B_T) set matching the ground truth.

$$BC = \frac{B_T}{n} * 100 \quad (1)$$

where n is number of bugs in the test data set.

This metric is same as the “Rank” metric used in [1, 20] which is defined as, the number of queries/bugs for which the relevant source files are retrieved with ranks rlow <= R <= rhigh where rlow = 1 and rhigh is configurable. We vary rhigh from 1 to 30, for the purposes of our experiments.

- *Average Precision, Recall and F1-Score:* For each bug in the test set, we calculate the traditional measures of precision, recall and F1-score.

As part of answering **RQ1** in Section 3.2.1 we discuss why the traditional measures of precision and recall are not good to compare the efficacy of the search techniques under investigation.

3.2 Effectiveness of the Indexing Techniques

In this study, we try to answer **RQ1** by evaluating the effectiveness of the three search techniques $\{$*MI:A*$\}$, $\{$*CI:A*$\}$ and $\{$*BI:A*$\}$. First, we present and discuss the efficacy of the three search techniques through the metrics P_A, R_A, $F1-Score_A$ and BC. Then, we statistically test whether the techniques are any better than chance.

3.2.1 Efficacy of the Search Techniques

For each bug in the test-set we perform the search over the indices 10 times, varying the size of the search results from 1 through 30. Figure 2 plots the average precision (P_A), recall (R_A) and F1-Score ($F1 - Score_A$) as line graphs across search techniques *MI:A*, *BI:A* and *CI:A*. Graphs in each columns plot the metrics P_A, R_A and $F1 - Score_A$ while each row represents these metrics calculated for each search technique.

In Figure 2, we observe that, across all subjects and search techniques, the average precision decreases and recall increases with increase in the size of the search result set. In contrast, the F-Score graphs shows a slight increase early on and then a steady decline. Beyond the search result size of 3, the precision decreases drastically as we return more results and the increase in recall does not compensate enough for the drop in precision. Usually, there are not too many files fixed in a bug (the mean being around 3 files per bug), so a large number of results are necessarily spurious when the size of the search results increases beyond a point. Interestingly, precision and recall for *Hadoop* are the highest amongst all techniques upto a search result size of 5. The F1-Score increases slightly as we move from a search result size of 1 to 3 and then drops steadily for all subjects.

The scale of the graphs indicate that the maximum values for precision, recall and F1-Score, across all subjects and techniques, are 0.3, 0.6 and 0.28 respectively. **The low increase in bug level recall indicates that text-search based techniques are not effective when the aim is to find all or most of the files to be modified for a bug.** Based on these observations about precision and recall, it seems that these are not good metrics to compare efficacy of various search techniques because all the numbers are very low. From an end user perspective, given a bug, text search should be used to find a file that can start the debugging investigation. So even if the search is able to return one correct result, it should be considered useful. Hence, we have chosen to use *Bug Coverage* as a comparison metric in rest of the paper.

Figure 3 plots the bug coverage as line graphs for each subject across the different sizes of search results and search techniques. We observe that, unlike F-Score, the values for bug coverage, across all subjects and techniques, increase with increase in the search result set size. For a maximum search result size of 30, at least one of the search techniques attains a minimum bug coverage of 55% for all the subjects. All the three techniques perform equally well for *Hadoop* with minimum bug coverage of 30% to a maximum of 70% across various search result sizes. In fact, *Hadoop* has similar bug coverage as an industrial project analyzed in [5], which reports a coverage of 68%. For *Birt*, searching over past bugs (*BI:A*) clearly outperformed the other two search techniques across all search result sizes. Interestingly, for *Datatools* the bug coverage remained constant beyond the search result size of 5 for the technique *BI:A*. A similar pattern is also observed in precision, recall and F1-Score for *Datatools* in Figure 2.

If we compare the different search techniques, *BI:A* and *CI:A* work better than *MI:A* across all our subjects. The extra effort in performing identifier splitting and other processing on actual code repository does not seem to yield any extra benefits as evident from the bug coverage observed for our subjects. Based on these results, for our subjects, none of the search techniques emerge as a clear winner (shown statistically in the later part of this section). However, the nature of the subject does impact the effectiveness of text-based search as observed for *Hadoop*.

3.2.2 Are the techniques any better than chance?

To objectively quantify that the search techniques are significantly better than chance, we compared their efficacy with that of a user who randomly selects source files from the code repository as the files to be fixed to resolve a bug. Suppose, there were n files in the repository when a bug was reported and f files were actually fixed to resolve the bug. Then, the hypergeometric distribution gives the probability, p, of getting at least x files that require a fix by choosing k files at random from the repository. [8]

[8] Think of the code repository as a bin of black and white balls, where the files that need fix for a bug resolution are considered to be white balls; rest

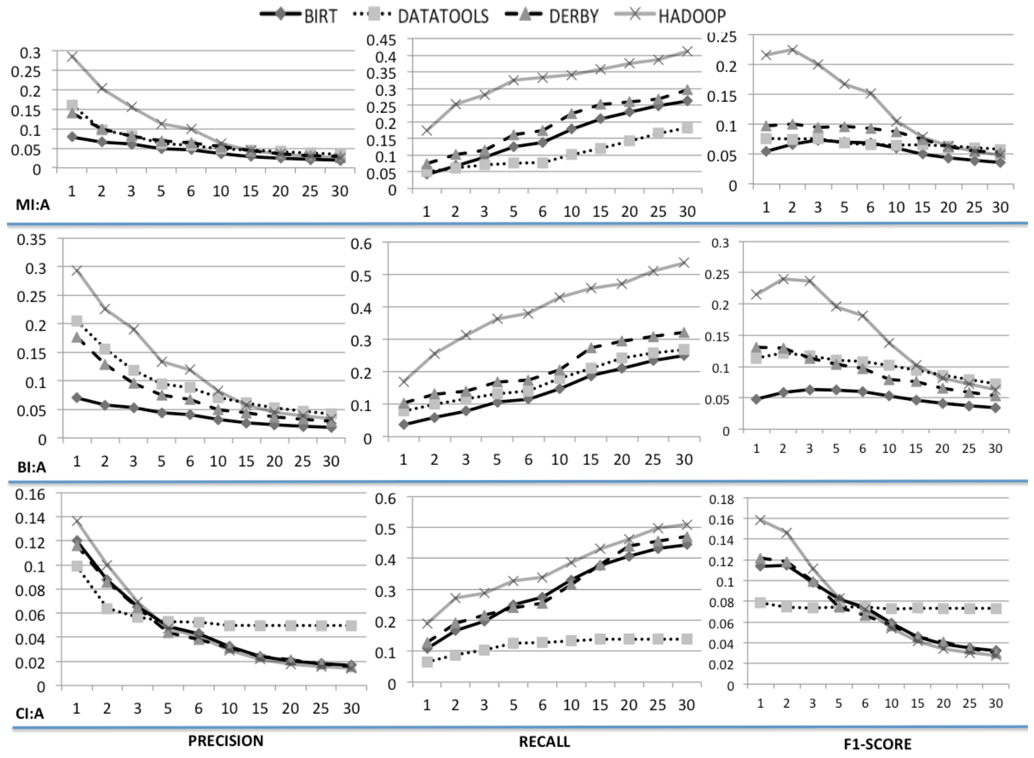

Figure 2. Average P_A (left), R_A (middle), and $F1 - Score_A$ (right) for MI:A, CI:A and BI:A across all subjects and search result sizes. X axis represents 10 iterations of the experiment with varying search result sizes. Y-axis represents the metrics value. Each line plots the scores across search result size for each subject.

$$p = 1 - \sum_{i=0}^{x-1} \frac{{}^{f}C_i {}^{n-f}C_{k-i}}{{}^{n}C_k}$$

Now, if a search technique returns x correct files in a set of k search results shown for a bug, then p gives the probability that one can get same or better results by drawing k files purely through chance. Thus, we use p (*p-value*) to test the null-hypothesis that a search technique is no better than chance. Further, we conduct the hypothesis test for all bugs (taken as queries) in our subject, which amounts to a case of multiple testing of hypotheses. To maintain an overall False Discovery Rate (FDR) of below 0.05, we lower the individual p-values during the testings by the Benjamini-Hochberg adjustment [6].

We tested the null-hypothesis for every bug in all our subjects separately for each of the three search techniques *BI:A*, *CI:A* and *MI:A*. Table 3.2.2 shows the percentage of bugs that recorded a p-value of less than 0.05 and that passed the FDR test for a search result size of 5. In general, the numbers look very close to that of bug coverage; indicating that even if one correct result is returned for a bug then the result is usually significant. Again, a high percentage of search results that are significant pass the FDR test too.

of the files are black balls. Now, the hypergeometric distribution gives the probability of choosing white balls without replacement.

Datatools is an exception where many of adjusted p-values miss the cut-off—in case of *MI:A*, almost half the cases that pass the significance test at 0.05 end up failing the FDR test. We traced the reason behind higher p-values to the fact that bugs in *Datatools* have higher number of files (f) associated with them. In two bugs as many as 491 files were found to be fixed. On an average across the subjects *Birt*, *Datatools* and *Derby*, 32% of the bugs analyzed for *BI:A*, 25% for *CI:A* and 21% for *MI:A* passed the significance test with $p < 0.05$. However for *Hadoop*, an average of 60% of bug reports passed across all techniques. Also, we record the average number of files per subject that can make the significance test fail (p = 0.05), in other words the number of files in the repository when the techniques break even with chance. The numbers range from 66 in *Derby* (*MI:A*) to 158 in *Datatools* (*CI:A*); suggesting that the techniques may not be applicable when the repository contains any lesser files.

Summary for RQ1

- No single search technique emerges as a clear winner.
- Size of code repository should be taken into account when deciding whether to use text search for fault localization or not. Break even score should be calculated. If bug triaging is already happening at a module level, then number of files in module should be considered to calculate break even.
- None of the search variants returns search recommendations with high precision, recall or bug coverage numbers and hence text search need to be applied in projects with realistic expectations.

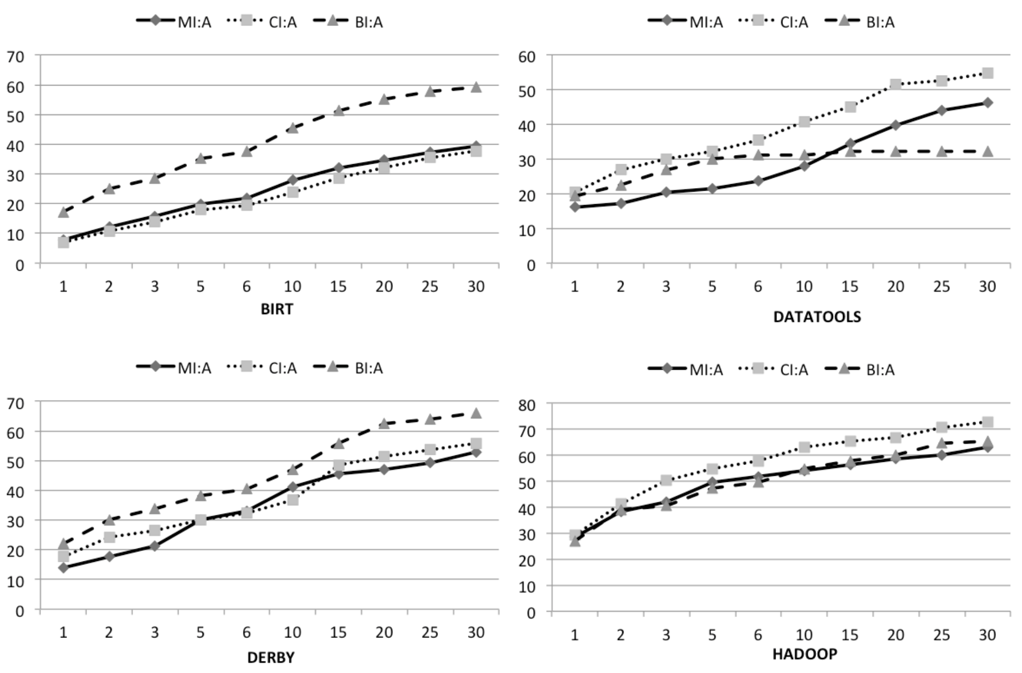

Figure 3. Bug Coverage for search techniques MI:A, BI:A and CI:A across the four subjects and search result sizes. X axis represents the various search result size. Y-axis represents BC (%). Each line plots the BC across search results for a given search technique.

Subject	BI:A		CI:A		MI:A	
	< 0.05	FDR	< 0.05	FDR	< 0.05	FDR
Birt	35.1	35	19.75	19.14	17.9	17.4
Datatools	30.1	23.6	31.18	20.43	20.43	9.67
Derby	37.5	34.5	30.1	28.6	29.4	26.5
Hadoop	47.4	41.4	54.9	51.1	41.6	41.4

Table 2. Percentage of Search Results that passed the significance test (p $<$ 0.05) and the FDR test

Subjects	MI:A & CI:A	MI:A & BI:A	CI:A & BI:A	MI:A & CI:A & BI:A
Birt	0.67	0.17	0.14	0.30
Datatools	0.67	-0.01	0.09	0.25
Derby	0.70	0.26	0.12	0.35
Hadoop	0.83	0.36	0.34	0.51

Table 3. Kappa Numbers for Search Result Size - 5

Subjects	Max (MI:A, CI:A, BI:A)	Combination			
		Rank Score	Norm Score	Aggregate Score	Sample Score
Birt	287	302	269	210	265
Datatools	30	44	40	32	42
Derby	52	58	57	47	60
Hadoop	73	74	78	70	76

Table 4. Comparison of Bug Coverage values between base and combination of techniques for search result size 5

3.3 Combining Search Techniques

In this study, we try to combine the results from the different search techniques evaluated in the previous subsection (what we call *base techniques* hereafter), in order to output more effective sets of recommendations; and hence answer **RQ2**. First, we check whether the results of the techniques are subsumed in one another. If so, then combining them may not be interesting at all. However, if the techniques turn out to be complementary, then we can design methods that synthesize a better set of recommendations by drawing from the results of the base techniques.

3.3.1 Kappa Analysis

For each subject, we perform Fleiss' Kappa [12] analysis to measure the degree of agreement (κ) amongst the three base techniques. Fleiss' κ is a statistical measure of inter-rater agreement or inter-annotator agreement for qualitative (categorical) items for any number of raters / annotators.

Each of our techniques map to an annotator (rater) and the categories are boolean–*Yes* indicates that the technique *covers* the bug (i.e., there is atleast one correct result returned), and *No* indicates that the technique does not cover the bug. With such an analogy, we setup a rater problem, where a technique (rater) rates a bug as *Yes* or *No*. Next, we compute the Kappa measure, κ, as:

$$\kappa = \frac{\bar{P} - \bar{P}_e}{1 - \bar{P}_e} \quad (2)$$

where $\bar{P} - \bar{P}_e$ gives the degree of agreement actually achieved above chance and $1 - \bar{P}_e$ gives the degree of agreement that is attainable above chance. $\kappa = 1$ indicates complete agreement between the raters (techniques) and $\kappa <= 0$ points to no agreement. For further details on how these are calculated please refer to [12].

As per the Kappa statistics (listed in Table 3.3.1), *MI:A* and *CI:A* turn out to be quite similar with at least 60% agreement. This is intuitive as both their indices are created from code files. Also, code comments are common in both code based index and meta index. However, both these techniques exhibit a very low rate of agreement with *BI:A*; 36% in the best case and -0.01% in the worst.

This analysis shows that bug based techniques and code or meta based techniques are significantly different from each other in terms of bugs that they address effectively. Hence, there lies a possibility improving the overall effectiveness of text-search (bug coverage) based fault localization by suitably combining these distinct techniques.

3.3.2 Heuristics for Combining Search Techniques

The aim behind combining results from the different search techniques is to improve the bug coverage. We apply the following heuristics to synthesize stronger recommendation sets by choosing from the results of the base techniques. (We evaluated these heuristics for all search result set sizes, but we present the data only for a search result size of 5 as a similar trend is observed for all other sizes.)

- *Rank based synthesis on score (RankScore)*: We rank all the search results across the three base techniques on the basis of their search similarity scores and choose the top X search results from this ranked set.
- *Rank based synthesis on normalized score (*NormScore*)*: Same as the above heuristic, the only difference being that the search scores are normalized (as a fraction of the maximum score returned by its query) before they are used for ranking.
- *Rank based synthesis on aggregate score (*AggregateScore*)*: For each file returned by a query, we sum up the similarity scores assigned to it by the different techniques. Next, we return the top X results from a ranked list prepared on the basis of such an aggregate score.
- *Sampling (*Sample*)*: We sample X search results for every query by picking the top 2*(X/5) search results from the results of $\{$*BI:A*$\}$ and $\{$*CI:A*$\}$, and the remaining X/5 results from $\{$*BI:A*$\}$. Since, $\{$*BI:A*$\}$ and $\{$*CI:A*$\}$ are the most complementary techniques (from Kappa Statistics), we sample a higher number of results from them–striving for greater diversity in the results.

Table 4 presents the average bug coverage of the recommendations when we apply the different heuristics to combine search results. *RankScore* technique works better than the best of the individual techniques for all four subjects. *NormScore* and *SampleScore* techniques work better in most cases except for *Birt*. *AggregateScore* is the worst performing heuristic across all subjects and search result size.

The improvement in bug coverage due to *RankScore* ranges from 1% (*Hadoop*) to 46% (*Datatools*).

Summary for RQ2

- *Search over code repository* and *Search over past bug repository* are complementary. So, they can be combined to increase the bug coverage.
- *RankScore* heuristic provides consistent improvement in bug coverage and is a feasible approach for combining search results stemming from different search techniques.

3.4 Impact of bug and code features on search techniques

In this study we explore the impact of different aspects of bug and code on our text-search system in order to answer **RQ3**. First, we evaluate whether biasing terms during query construction improves the efficacy of search. Then, we inspect how the search performance is impacted by different physical aspects of bug reports (size, number of code terms etc.) and source code (lines of code, length of comments etc.). Finally, we study the impact of external factors such as the vocabulary overlap between bug queries and search indices, and the number of files fixed per bug.

3.4.1 Biasing terms during query construction

The TF-IDF scheme assigns a weight to a term (or word) such that it indicates the relative importance of the term in a document or a query in the context of the search index. However, when the number of words in a query becomes large, it may so happen that certain key terms do not get adequately high weights in the query vector. In such a scenario, if we have a good idea about the words in the bug report that may be important from a search perspective, then we can adjust the default TF-IDF weights in a query vector to direct greater focus on those *keywords*.

Table 3.4.1 shows the size of queries created (in base techniques) for each of our subjects–*low*, *high*, and *mean*. It also lists the average number of words in the title—title-words, and the average number of code terms present in these queries—code-words. Note, that we remove the stop words and Java specific keywords before calculating the queries. The mean query sizes vary from 20 for *Hadoop* to 157 for *Birt*. The number of words in title and the number of code terms are very less (4 to 17). To counter the effect of large query size, we evaluate the two query construction strategies—giving boost to title words, (*TB*), and giving boost to code words, (*CB*); we compare their effectiveness with that of the simple–All (A) strategy.

Bug coverage results of the different search techniques formed with *TB* and *CB*, are compared with those of the base techniques in the Figure 4. The graph is plotted only for search result size of 5. X- axis represents the subjects and y - axis represents the bug coverage (%). Per subject there are 9 vertical bars. The first three bars corresponds to the *MI* index, the next three corresponds to *BI* index and the last three are for *CI* index. This segregation is indicated by a vertical line in the figure. Further within the bars corresponding to

Subject	Low	High	Mean	Title	Code
Birt	2	311	157	9	9
Datatools	18	94	56	4	17
Derby	9	234	67	6.5	19
Hadoop	3	130	20	5	7

Table 5. Query Size (number of words)

Subjects	Index	Query Construction					
		A		*TB*		*CB*	
		CW	TW	CW	TW	CW	TW
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Birt	MI	75.2	28.7	78.7	41.8	80.6	31.1
	BI	62.6	22.7	64.2	29	66.7	24.3
	CI	73.2	27.9	74.8	40.3	79.2	29.5
Datatools	MI	65.7	20.8	71.7	34.7	77.0	24.7
	BI	47.6	13.8	54	26.0	55.7	17.6
	CI	64.6	20.2	70.2	31.7	73.6	20.5
Derby	MI	57.7	16.5	65.1	23.9	68.9	20.3
	BI	49.4	14.6	53.1	26.8	59.5	19.1
	CI	57.9	16.3	64.5	23.4	68.8	19.7
Hadoop	MI	65.5	57.8	65.4	66	69.3	59.3
	BI	58.2	59.9	59.0	75.2	67.1	64.0
	CI	63.1	57.6	63.3	65.2	67.9	59.1

Table 6. Percentage of significant Code Words (CW) and Title Words (TW)

these indices, the first one corresponds to base technique *A* (empty bar), the second corresponds to title boost *TB* (vertical line bar) and third one is code boost - *CB* (horizontal line bar).

As evident from the plot, between *TB* and *CB*, boosting the title words during the query construction (*TB*) had positive impact on bug coverage across all indices and subjects except for *Hadoop*. The maximum gain in coverage was observed for *Derby* (11%) for *BI* index. However, boosting code words (*CB*) showed more mixed results. It only provided an increased bug coverage for *Derby* over the base *BI* and *MI* techniques. It also improved the bug coverage for *Hadoop* over the *MI* technique. Overall *TB* technique seems to outperform *CB* techniques for the subjects and indices considered.

Now, we investigate the reason why title-boost proves to be beneficial and not code-boost. For every search query that is run, our Lucene-based search system returns the set of query terms that are significant in computing the search-similarity scores. Table 6 lists the percentage of title-words (TW) and code-words (CW) that feature in the set of significant query terms, across the subjects, when different search techniques are applied. The rows represent each indexing technique (across all subjects) and the columns represent the query construction strategies. Columns (3) and (4) correspond to All (*A*) , columns (5) and (6) to Title-Boost (*TB*), and columns (7) and (8) to Code-Boost(*CB*) strategies.

For *Hadoop*, the queries used in the base techniques contain more than 57% of the significant title words, while the queries of the other three subjects have less of such words (22% for *Birt*, 13% for Datatools, 14% for Derby). When title boost (*TB*) is applied the percentage of significant title

Subjects	Unique Terms			Intersect	
	MI	CI	Bugs	Bugs-MI	Bugs-CI
Birt	41621	148451	6460	2257 (35%)	2778 (43%)
Datatools	24634	70763	1818	1098 (60%)	1152 (63%)
Derby	28315	65913	65797	1940 (50%)	2021 (51%)
Hadoop	14070	55285	1610	1020 (63%)	1131 (70%)

Table 7. Size of Repository Index and Overlap with Bug Test Set

words in queries almost doubles for *Datatools* (from 13.8% in *MI:A* to 26%) resulting in a increase of bug-coverage. However for *Hadoop*, further boosting of title words does not actively contribute toward increases in bug coverage because most of the keywords in the title are already considered to be significant even in the base techniques. Unlike title words, the percentage of significant code words are much higher (47% - 75%) for all subjects in the base techniques—*MI:A*, *CI:A* and *BI:A*. Also, when *TB* is applied, the code words in the title also get a boost (column (5) > column(3)).

To summarize, **giving preference to title-words in the query helps to improve the effectiveness of the search**. On average when compared to the base techniques across all indices, title boost *TB* provided 1.6% increase in bug coverage for *Birt*, 3.9% for *Datatools* and 7.8% for *Derby*. For *Hadoop*, this did not work as already title words were well represented in the significant query terms.

For each bug report under analysis, we collect the following features: *Total number of words*, *Number of code words* and *Number of title words*. Further, we note the *Search-Similarity score* returned by the search engine as we execute queries. Next, we compute *Spearman's rank correlation coefficient* between the features and whether the search technique returns a correct or an incorrect match for that bug, i.e., coverage at the bug level. We find the *Search-Similarity score* to be positively correlated across almost all subjects (except Hadoop) and techniques. This confirms the intuition that the likelihood of suggesting a correct match is high if the Search-Similarity score is high. The number of code words is positively correlated with coverage in 5 of the 12 cases. So, there is some evidence that code words in bug description can improve search performance; however we cannot be conclusive about this result. The total number of words in a bug report is negatively correlated in some cases–perhaps indicating that the search techniques are not able to deal with too much noise in the bug descriptions.

3.4.2 Impact of Subject Level Features

The high efficacy of all search variants on *Hadoop* (Figure 3) made us investigate if there are specific subject level features that bias text-based search. We analyzed two factors:

- Do the techniques work well only when the number of files to be fixed associated with bug reports is large? Figure 5 plots the spread of number of files fixed for each bug in our test suite—per subject. Except for a

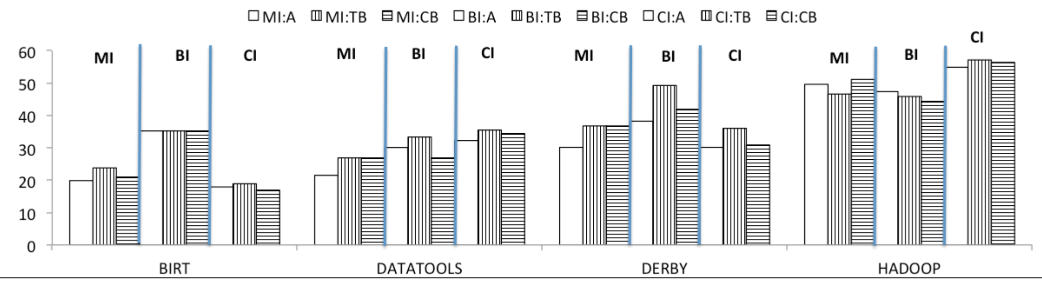

Figure 4. Bug coverage (BC %) across the base techniques and boost techniques are plotted for all subject for search result size of 5. X axis represents the subjects. Y axis represents the bug coverage.

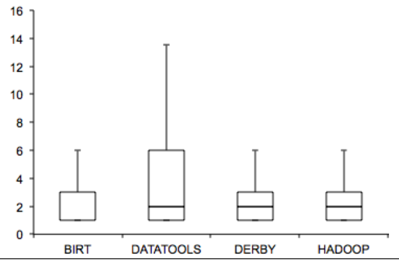

Figure 5. Spread of files modified per Bug per subject in the oracle. The red stars indicate outliers. The bottom and top whiskers show the low 25 and top 25 percentile. The box shows the spread for remaining 50 percentile.

few outliers (ranging from 4% for *Datatools* to 10% for *Hadoop*)), 75% of the bugs across subjects require 6 files or less to be modified and remaining 25% require 15 files or less . The wider spread of upto 15 files is due to *Datatools*, which also impacts the FDR significance test for this subject (as discussed in Section 3.2). The spread for *Hadoop* is very much similar to *Derby* or *Birt*, hence the measure of number of files fixed per bug does not seem to have any impact on the text based search.

- Do the code search techniques work well only when there is a high vocabulary overlap between the code index and the bugs analyzed? Table 7 lists the size of the two variants of code index *CI*, *MI*. Column 4 gives the number of unique terms in the bug queries created from the test suite. Column 6 and 7 gives the absolute and percentage of intersection between words in the test suite (bugs) and the two code indices. The overlap is highest in *Hadoop*—63% for *CI* and 70% for *MI*. This validates the fact that these techniques work the best for *Hadoop* (as observed in Figure 3). The overlap is lowest for *Birt*—manifested in the fact that we note the lowest bug coverage for *Birt* when we search over the code index (*CI*). Thus, across subjects, **the vocabulary overlap between code index and bug reports impacts search efficacy**. Again, we note that identifier splitting does not lead to an increase in vocabulary match—note the percentage overlap across columns 5 and 6 in table 7. The bug to index vocabulary match is higher in the *CI* technique (where code is not processed) than the *MI* technique (where the code is pre-processed with identifier splitting).

Summary for RQ3

- Boosting the title words as part of query construction helps in increasing the bug covergae. Minimum of 1.65% in *Birt*, and maximum of 7.8% in *Derby*.
- Too many words in bug description can negatively impact search accuracy.
- There is some evidence that presence of code words can help search. However, we are still not conclusive.
- Search similarity score seems to be the only feature, which is almost always significantly correlated to a correct or incorrect search match.

4. Related Work

Traditionally, research on fault localization always meant application of some program analysis or debugging technique. Different variants of program slicing [4] have been tested on their efficiency of localizing faults. However, the *slicing criterion* (the point of interest in the program where analysis can start) may not be always clear from a description of a reported bug; thus slicing techniques cannot be immediately applied. Statistical debugging or spectra-based fault localization [13] evaluates various program spectra and pass/fail status of test cases in order to compute the risk of containing a fault for each program entity. Such techniques need a large number of passing test cases to be effective, which may not be always available in practice. Delta debugging [23] instruments the test environment such that it is possible to systematically make the input to a failing test case smaller. Mutation based approaches [21] modify the program state or control flow such that failing test cases can pass. However, they are not very scalable since the search space of program states can become really large and can only be applied to limited types of faults.

Hipikat [8] applies information retrieval to recommend existing software development artifacts (e.g., change tasks, design documents, source files) in context of a task being performed. *PROMESIR* [11] shows that Latent Semantic Indexing (LSI) and scenario based probablisitic ranking (SPR) methods can be effective to locate features (formulated from title and description of bugs) within source code. Lutkins et al. [1] show that querying a Latent Dirichlet Allocation (LDA) model built from the source code can outperform LSI-based methods. Rao and Kak [20] evaluate five generic text models—Unigram Model, Vector Space Model, Latent Semantic Analysis, LDA and Cluster Based Decision Making with respect to their effectiveness in localizing bugs of the iBUGS dataset [9]. *DebugAdvisor* [5] can launch fat semantic queries (comprising of bug description, debugger output, logs etc.) over software repositories that aggregate data from diverse systems such as version control systems, debugging sessions and bug databases.

To the best of our knowledge, there exist no prior work that compares and contrasts the approaches of searching over source code and past bug reports; and the benefits of combining them.

5. Conclusion

In this paper, we study the efficacy of different text-search approaches in bug localization through an empirical evaluation on four open source subjects. Specifically, we examine techniques that directly search the code repository and those that search over a historical collection of bug reports. Overall, no technique comes out as significantly better over the other; for a search result set size of 5, code repository search as well as bug repository search yielded a bug coverage varying from 20 to 60% across our study subjects. Doing any pre-processing of the code to split identifiers into words did not yield benefits. However, we find that these techniques are complementary; we measure an improvement in the bug coverage (1% - 46%) when the techniques are applied in tandem. Favoring words in title help in producing better recommendations in most cases. All experimental data (bugs, truth set, search results) are available here [9].

[9] `https://sites.google.com/site/searchbugs/`

References

[1] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, 2008.

[2] L. P. . Enslen, E. Hill and K. Vijay-Shanker. Mining Source Code to Automatically Split Identifiers for Software Analysis. In *Working Conference on Mining Software Repositories, MSR*, pages 71–80, 2009.

[3] H. Agrawal, R. Demillo, and E. Spafford. Debugging with dynamic slicing and backtracking. *Software: Practice and Experience*, 23(6):589–616, 1993.

[4] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 143–151. IEEE, 1995.

[5] B. Ashok, J. Joy, H. Liang, S. Rajamani, G. Srinivasa, and V. Vangala. DebugAdvisor: A Recommender System for Debugging. In *Proceedings of the the 7th joint meeting of the ESEC/FSE*, pages 373–382. ACM, 2009.

[6] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 289–300, 1995.

[7] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, pages 34–44, 2009.

[8] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Trans. Softw. Eng.*, 31:446–465, 2005.

[9] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 433–436. ACM, 2007.

[10] R. A. DeMillo, H. Pan, and E. H. Spafford. Critical slicing for software fault localization. pages 121–134, Jan. 1996.

[11] Y. Denys Poshyvanyk et al. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, pages 420–432, 2007.

[12] J. F. et al. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.

[13] J. Jones and M. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282. ACM, 2005.

[14] B. Korel and J. Rilling. Application of dynamic slicing in program debugging. In *Proceedings of the 3rd International Workshop on Automatic Debugging*, May 1997.

[15] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26, 2005.

[16] B. Lientz, E. Swanson, and G. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, 1978.

[17] C. Manning, P. Raghavan, and H. Schutze. Introduction to information retrieval. Cambridge University Press 2008.

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

[19] J. McKEE. Maintenance as a function of design. In *Proceedings of the National Computer Conference and Exposition*, pages 187–193. ACM, 1984.

[20] S. Rao and A. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceeding of the 8th working conference on Mining software repositories*, pages 43–52. ACM, 2011.

[21] C. D. Sterling and R. A. Olsson. Automated bug isolation via program chipping. 37(10):1061–1086, Aug. 2007.

[22] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

[23] A. Zeller. Isolating cause-effect chains from computer programs. *ACM SIGSOFT Software Engineering Notes*, 27(6): 1–10, 2002.

[24] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.