On the Use of Stack Traces to Improve Text Retrieval-based Bug Localization

Laura Moreno[1], John Joseph Treadway[2], Andrian Marcus[1], Wuwei Shen[2]

Department of Computer Science
The University of Texas at Dallas
Richardson, TX, USA
lmorenoc@utdallas.edu, amarcus@utdallas.edu

Department of Computer Science
Western Michigan University
Kalamazoo, MI, USA
john.j.treadway@wmich.edu, wuwei.shen@wmich.edu

***Abstract*—Many bug localization techniques rely on Text Retrieval (TR) models. The most successful approaches have been proven to be the ones combining TR techniques with static analysis, dynamic analysis, and/or software repositories information. Dynamic software analysis and software repositories mining bring a significant overhead, as they require instrumenting and executing the software, and analyzing large amounts of data, respectively. We propose a new static technique, named Lobster (LOcating Bugs using Stack Traces and tExt Retrieval), which is meant to improve TR-based bug localization without the overhead associated with dynamic analysis and repository mining. Specifically, we use the stack traces submitted in a bug report to compute the similarity between their code elements and the source code of a software system. We combine the stack trace based similarity and the textual similarity provided by TR techniques to retrieve code elements relevant to bug reports. We empirically evaluated Lobster using 155 bug reports containing stack traces from 14 open source software systems. We used Lucene, an optimized version of VSM, as baseline of comparison. The results show that, in average, Lobster improves or maintains the effectiveness of Lucene-based bug localization in 82% of the cases.**

***Keywords*—bug localization; stack traces; static analysis; text retrieval.**

I. Introduction

Identifying where to make changes in a software system in response to a change request, also known as *concept location* [4, 20], is an essential task during software evolution. More than two decades of research in this area have focused on providing tool support to developers, taking advantage of textual, static, and/or dynamic information extracted from different software artifacts [10, 16]. *Bug localization* is an instance of concept location, where the change request is expressed as a bug report and the end goal is to change the existing code to correct an undesired behavior of the software. Existing studies [17] revealed that bug reports and source code (i.e., identifiers and comments) share significant parts of their vocabulary. Text Retrieval (TR) based bug localization takes advantage of this fact and uses the textual similarity between a bug report and the source code of a software system to identify the code elements related to the bug. The most successful feature/bug localization approaches have shown to be the ones combining different analyses and sources of information (e.g., [1, 2, 6-8, 11, 15, 18, 19, 21, 23, 25, 27-29]). These approaches usually utilize TR techniques in conjunction with static/dynamic analysis or Software Repository Mining (MSR).

In general, bug localization approaches based on dynamic analysis and MSR require more effort, data, and processing time than the approaches solely based on static analysis. For example, approaches based on dynamic information (e.g., [11, 15, 19]) require instrumenting the software in order to collect the desired data, such as, runtime execution traces. MSR-based approaches (e.g., [7, 29]) require collecting and analyzing historical data of software (e.g., issue trackers and control version systems), such as, similar bug reports or the change history of the code. Both, the data extraction and its analysis are often time consuming for large repositories and subject to data quality issues. The difficulties associated with utilizing this extra information (i.e., dynamic or historic) during bug localization make these approaches often impractical. On the other hand, static analysis based approaches (e.g., [2, 6, 14, 18, 25, 28]) only require the version of the software system to be modified in order to extract structural information, such as, code dependencies or syntax information. Given this context, our research focuses on improving static TR-based bug localization approaches by using additional information, like the one extracted from execution traces, but without the added cost.

Bug localization approaches based on TR leverage the textual similarity between the bug descriptions and the source code. In short, these approaches take as input a *query* built from the textual description of a bug report and use it to retrieve similar documents from a *corpus*, built from the source code of the software system at hand. The more similar a document in the corpus is to a bug report, the higher the chances it will have to be changed to fix the bug. Bug reports, however, consist of more than text—they sometimes contain fragments of code, stack traces, etc. A recent study [24] showed that about 8% (12,947 out of 161,500) of the bug reports from Eclipse (submitted until October 2006) contain at least one stack trace. Stack traces have been shown to be useful in bug localization [7, 27, 29]. The aforementioned study [24] also reported that about 60% of the bug reports that contain stack traces and whose code changes could be identified (i.e., 3,940) were fixed in one of the stack frames submitted in the report. This means that in 60% of cases, the stack trace submitted in a bug report contains (at least) one code element that needs to be modified in order to fix the bug at hand. We conjecture that in the other cases the code elements to be changed are near the ones present in the stack traces (in terms of dependency graph neighborhood). In consequence, we argue that the stack trace information can be used for improving TR-based bug localization approaches.

This improvement would come at a very low cost in all the cases where stack traces are present.

In this paper, we present Lobster (LOcating Bugs using Stack Traces and tExt Retrieval), a novel approach that leverages: (i) the text of bug reports and source code of a software system, (ii) the stack traces submitted in bug reports; and (iii) the source code structure, in order to identify code relevant to a bug report. For this, Lobster uses a measure that combines the *textual similarity* between a bug report and a code element, and the *structural similarity* between the stack trace submitted in the bug report and the code elements. The structural similarity is given by the minimum distance between the elements in the stack trace and the target code element. We evaluate Lobster on a dataset consisting of 155 bug reports from 14 open source projects, using Lucene as a baseline for comparison. The results show that Lobster improves the effectiveness of TR-based bug localization in 52.6% (in average) of the cases when compared to Lucene. In addition, in 29.0% (in average) of the cases, Lobster preserves Lucene's effectiveness.

The rest of the paper is organized as follows. We present Lobster, our new bug localization approach, in Section II. The empirical evaluation of the approach and the results are described in Sections III and IV, respectively. In Section V, we summarize how this work relates to other research. We draw conclusions and describe future work in Section VI.

II. COMBINING TEXT RETRIEVAL AND STACK TRACES TO LOCATE BUGS

As mentioned before, bug reports occasionally contain one or more stack traces that display the sequence of nested functions that were being executed up to the point where the reported bug occurred. While the buggy functions are not always present in the stack trace [24], we presume that they are structurally related to the functions in the stack trace (i.e., there is a direct or indirect dependency from the erroneous function to the ones in the stack trace). Our new bug localization technique relies on this assumption.

Lobster (LOcating Bugs using Stack Traces and tExt Retrieval) is a technique supporting bug localization that combines TR and static software analysis by leveraging the stack traces reported in a bug report. Lobster considers the vocabulary of a bug report and the source code of a software system in order to find *textually similar* code elements to the bug at hand, as provided by traditional TR-based bug localization. In addition, Lobster uses the stack traces found in the bug report and a dependency graph of the software to find *structurally similar* code elements to the ones in the stack trace. In this way, given a bug report and a code element, their combined *total similarity* is given by the linear combination of the textual similarity and the structural similarity between them. The higher the total similarity is, the more relevant to the bug report the code element is. The expected improvements come from that fact that the total similarity measure will penalize code elements that have (accidental) high textual similarity to the bug description but are not structurally related to those in the stack trace, hence unlikely to contribute to the reported buggy behavior. In the following subsections we describe in details the similarity measures.

A. Textual Similarity

We define the textual similarity between a bug report and a code element *e* as:

$$\text{sim}_{\text{textual}}(bugReport, e) = \text{score}_{\text{TR}}(bugReport, e)$$

where the score function is given by any TR technique that provides a relevance measure between documents, such as, VSM [22], LSI [9], or LDA [5]. For example, in VSM, documents are represented as vectors of words (i.e., the vector space). The similarity of a document to a query is given by the cosine between the vectors representing the document and the query. VSM has been widely used in TR-based bug localization [10, 16]. In our implementation of Lobster used for empirical evaluation (see Section III), the textual similarity is given by Lucene [13], a TR model that has been shown to outperform techniques like LSI in bug localization [17] by combining VSM and a Boolean model.

B. Structural Similarity

Given a stack trace and a code element *e*, we define the structural similarity between them in terms of the minimum distance between the code elements in the stack trace and *e*. The computation of such distance is based on the program dependence graph [6] of the software system, i.e., a directed graph where each node represents a distinct code element of the system, and an edge from one node to another represents control or data flow. Let us assume, for example, that classes are the code elements to be analyzed. In such case, each node in the program dependence graph represents a class, and an edge from node c_1 to node c_2 could be an invocation of a method of the class c_2 by a method of the class c_1.

We define the distance between a stack trace and a code element *e* as the minimum shortest path between each code element listed in the stack trace and the target code element *e* and vice versa, as follows:

$$\text{dist}(stackTrace, e) = \min\begin{pmatrix} \forall_{d \in stackTrace}\text{shortestPath}(d, e) \\ \cup \forall_{d \in stackTrace}\text{shortestPath}(e, d) \end{pmatrix}$$

If no path exists between two code elements, the shortest path between them is equal to infinite. Note that if the code element *e* is listed in the stack trace, the distance between them is equal to zero (0).

We define the structural similarity between a stack trace and a code element *e* as the complement of the normalized distance between them, as follows:

$$\text{sim}_{\text{struct}}(stackTrace, e) = 1 - \frac{\min(\text{dist}(stackTrace, e), \lambda)}{\lambda},$$

with $\lambda \geq 1$. This parameter is a threshold defining the maximum considered distance. Note that the structural similarity values are in the range [0,1], where one (1) represents maximum similarity and zero (0) represents no similarity. For example, let us assume that $\lambda = 1$. In this case, the structural similarity can take only two values: one (1) when the code element *e* is listed in the stack trace (i.e., $\text{dist}(stackTrace, e) = 0$), or zero (0), otherwise. When $\lambda = 2$, the structural similarity can take three values: one (1) when the

TABLE I. SYSTEMS USED IN THE EVALUATION AND THEIR PROPERTIES

System	Version	# of Bug Reports	# of Bug Reports with Stack Traces	# of Classes
ArgoUML[a]	0.22	91	20	1,635
BookKeeper[b]	4.1.0	43	8	587
Derby[b]	10.7.1.1	33	10	3,040
	10.9.1.0	96	26	3,132
Hibernate[b]	3.5.0b2	21	3	4,037
JabRef[a]	2.6	39	3	856
jEdit[a]	4.3	150	8	1,014
Lucene[b]	4.0	35	5	4,317
Mahout[b]	0.8	30	7	3,260
muCommander[a]	0.8.5	92	4	1,443
OpenJPA[b]	2.0.1	35	6	4,438
	2.2.0	18	4	4,955
Pig[b]	0.8.0	85	17	2,095
	0.11.1	48	12	2,506
Solr[b]	4.4.0	55	3	1,863
Tika[b]	1.3	23	3	582
ZooKeeper[b]	3.4.5	80	16	752
Total		*974*	*155*	*40,512*

[a.] Part of a benchmark in TR [10]
[b.] Data automatically extracted from JIRA

code element e is listed in the stack trace; 0.5 when the code element e is directly called by or calls an element in the stack trace (i.e., $\text{dist}(\mathit{stackTrace}, e) = 1$); or zero (0), otherwise. In other words, λ indicates how far (in the dependence graph) from the stack trace's elements a code element can be to be considered similar. For example, $\lambda = 1$ means that only the elements in the stack trace are similar, whereas $\lambda = 2$ also scores the neighbors in the dependency graph. $\lambda = 3$ will also score the neighbors of neighbors, and so on. We study the effects of varying λ in the next section.

C. *Total Similarity*

Finally, we define the *total similarity* between a bug report and a code element e in the software as a linear combination between their textual and structural similarities, as follows:

$$\text{sim}(\mathit{bugReport}, e) = (1-\alpha) * \text{sim}_{\text{textual}}(\mathit{bugReport}, e) + \alpha * \text{sim}_{\text{struct}}(\text{getStackTrace}(\mathit{bugReport}), e)$$

where the function `getStackTrace` extracts the stack traces from the bug report, and the parameter $\alpha \in [0,1]$ adjusts the weights of the textual and the structural similarities within the total similarity. Note that when $\alpha = 0$, the structural similarity is disregarded and the total similarity is equal to the textual similarity (i.e., the traditional TR approach), whereas when $\alpha = 1$, the structural similarity is disregarded and the total similarity is equal to the structural similarity. We study the effects of varying α in the next section.

III. EVALUATION

We conducted an empirical study to evaluate the performance of Lobster in supporting bug localization. Our study is designed to determine whether the proposed total similarity improves the performance of TR-based bug localization. In addition, we investigate the effect of the λ and α parameters on Lobster's performance.

A. *Subject Systems*

To evaluate our approach, we use data from 17 versions of 14 open source software systems written in Java, which are summarized in Table I. These systems vary in size and problem domain. ArgoUML is a UML diagramming tool; BookKeeper is a record stream logger; Derby is a relational database management system; Hibernate is an object-relational mapping library for Java; JabRef is a reference manager; jEdit is a multi-platform text editor; Lucene is a widely used information retrieval library; Mahout is scalable machine learning library; muCommander is a cross-platform file manager; Pig is a platform for creating MapReduce programs; Solr is a scalable search platform; Tika is a structure data detector and extractor toolkit; and ZooKeeper is a centralized service for maintaining distributed applications. Two versions of Derby, OpenJPA and Pig were considered in the study, whereas for the other systems only one version of each one was considered. The data of ArgoUML, JabRef, JEdit and muCommander is part of a benchmark data set widely used for evaluating TR-techniques in feature location research [10]. The other data sets corresponds to Apache projects, whose data was automatically extracted by analyzing the issues of each project from the Apache's JIRA issue tracker (https://issues.apache.org/jira/). In each case, we extracted the issues whose resolution was marked as "fixed" or "closed" and whose patch files were available (i.e., attached to the issue). These patch files contain detailed information of the changes made by developers in response to an issue. We identified the classes deleted and modified in the patch files by using regular expressions. The source code of each project was obtained from the Apache Archives (https://archive.apache.org/).

The complete data set consists of 974 bug reports, of which 155 (15.9%) contain stack traces (see Table I). Each bug report in the data set points to its *patched classes*, i.e., the code classes that were modified to fix the bug. Since Lobster only affects the retrieval of the bug reports containing stack traces, we focus our evaluation on this subset (i.e., 155 bug reports). We perform all the analyses at *class level*, meaning that the documents in the corpus and the nodes (i.e., code elements) in the program dependence graph correspond to the Java classes of the subject systems. Investigating other document granularity (e.g., method level) and systems in other languages (e.g., C++, C#, etc.) is subject of future work.

The complete data used in the evaluation is available in an online Appendix at: http://tinyurl.com/lobster14.

B. *TR Technique*

As mentioned in Section II, several TR techniques have been used in bug localization. In principle, any TR technique providing a relevance score of a document with respect to a query (e.g., VSM, LSI, LDA, LM, etc.) could be used to compute the textual similarity in our approach. We use Apache Lucene (http://lucene.apache.org/) for the evaluation of our approach with its default similarity measure. Future work will study the effect of the structural similarity when used in conjunction with other TR techniques and configurations.

C. *Corpus Creation*

We represent the documents (i.e., bug reports and code classes) of each system in our data set as "bags of words", a common approach when using TR techniques, such as Lucene, for bug localization. To this end, we create one document for each bug report and class in each subject system. When analyzing bug reports, we extract the text from their title and description, whereas for code classes, we extract the text from their identifiers, comments, and literals.

We normalize the text of all documents by following common text preprocessing techniques [10, 16] in TR-based bug localization. First, we split identifiers according to Camel case notation and by removing special characters, but we also keep the original ones. Next, we remove common English stop words and programming keywords (see our online Appendix). Finally, we stem the words using the Porter stemmer.

D. *Program Dependence Graph Extraction*

For the purpose of this evaluation we decided to implement in Lobster a call graph (i.e., a simplified program dependence graph). In order to build such call graph, Lobster uses the *java-callgraph* suite (https://github.com/gousiosg/java-callgraph), a set of utilities for generating static and dynamic call graphs from Java systems. Specifically, we use the *javacg-static* program that takes as input a jar file and generates a call graph of its classes and methods through software static analysis. The output of such an analysis is a tabulated file containing caller-callee relationships of classes and methods. Since our evaluation is at class level, we only consider the relationships between classes.

E. *Stack Trace Identification*

Similar to previous work [3], we use regular expressions to extract the stack traces from bug reports. In general, a stack trace consists of the stack frames that were active when an exception, error, or assertion violation occurred during the execution of a program. Our focus is on the classes listed in the stack frames. We use a regular expression derived from the next abstraction to identify such classes:

```
[packageName]?[className].[methodName](
[filename].java:[lineNumber]|[unknown source|native method])
```

F. *Methodology*

In order to assess the performance of Lobster, we use it on the 155 bug reports of the data set that contain stack traces (see Table I). For each bug report, we retrieve the list of relevant code elements by varying the distance threshold λ from 1 to 3 and the weight of the structural similarity α from 0 to 1 with steps of 0.1. Note that $\alpha = 0$ means disregarding the structural similarity, i.e., using Lucene solely (the traditional TR approach). We use this setting (i.e., $\alpha = 0$) as the baseline to compare the performance of the proposed approach against. It is common practice in the research on concept and feature location [10] to use such a baseline. Comparing against hybrid approaches (which include, for example, data from execution traces) is not done due to the practical differences between the static and dynamic techniques mentioned in Section I. Alternative baselines could be the MSR-based approach proposed by Zhou et al. [29], the term boosting-based approach proposed by Bassett et al. [2], or the clustering-based approach proposed by Scanniello et al. [23]. As with dynamic and hybrid techniques, some of these require additional, potentially costly steps, such as, clustering the software system [23] or mining issue trackers and control version systems to analyze similar bug reports and their corresponding code changes [29]. In future work we will consider such additional baselines and perform a thorough cost-benefits analysis of the additional information utilized by them vs. the existing stack traces.

Based on previous work on bug and feature location [10], we use the following measures in the evaluation:

- *Effectiveness* is a measure that approximates developers' effort during the localization task. This effort is measured as the number of documents that need to be investigated by a developer before finding a document relevant to her query. The effectiveness is defined as the best rank obtained by the set of documents R_q relevant to a query q within the list of retrieved documents, when sorted in descendent order with respect to the similarity between the query and each document in the corpus:

$$\text{effectiveness}(q) = \min\left(\forall_{r\in R_q} \text{rank}(r)\right)$$

  The lower the effectiveness value is, the more effective the bug localization technique is.

- *Mean Reciprocal Rank* (MRR) is a statistic that measures the quality of the ranking of a retrieval approach in a search task, by capturing how early the relevant document to a query is retrieved. MRR is given by the average between the reciprocal effectiveness of a set of queries Q:

$$\text{MRR}(Q) = \frac{1}{|Q|}\sum_{\forall q\in Q}\frac{1}{\text{effectiveness}(q)}$$

  The higher the MRR value is, the higher the ranking quality of the bug localization approach is. Note that this measure correlates to the effectiveness—while MRR is used for collections of queries, the effectiveness provides fine grained analysis of individual queries.

- *Mean Average Precision* (MAP) is a measure of the accuracy of a retrieval approach based on the average precision of each query q in the set Q. Given the set of documents R_q relevant to the query q, the average precision is computed as the average of the precision values at the resulting rank of each document $r \in R_q$. MAP is the mean of the average precision of the set of queries Q, defined as follows:

$$\text{MAP}(Q) = \frac{1}{|Q|}\sum_{\forall q\in Q}\frac{1}{|R_q|}\sum_{\forall r\in R_q}\text{precision}(\text{rank}(r))$$

To establish if the difference between an evaluation measure obtained by applying two different configurations of Lobster is statistically significant, we use the Wilcoxon test [26], a non-parametric test for comparing paired samples whose distributions are not assumed to follow a normal distribution (which is our case). For more than two samples, we used the Friedman test [26]. We use the Cliff's delta [12] to quantify the magnitude of the difference in the evaluation measures. The magnitude is assessed using the guidelines provided in previous research [12], i.e., *negligible* for $|d| < 0.147$, *small* for $0.147 \le |d| < 0.33$, *medium* for $0.33 \le |d| < 0.474$, and *large* otherwise.

TABLE II. BUG REPORTS (BRS) AND THEIR PROPERTIES IN TERMS OF STACK TRACES (STS) AND PATCHED CLASSES (PCS)

System	# of BRs with STs	# of PCs[a]	# of PCs with[b]					# of BRs with PCs with[b]	
			dist(*ST*,*PC*)=0	dist(*ST*,*PC*)=1	dist(*ST*,*PC*)=2	3≤dist(*ST*,*PC*) ≤7	dist(*ST*,*PC*)=∞	dist(*ST*,*PC*)=0	dist(*ST*,*PC*)≠∞
ArgoUML 0.22	20	33 (1.7)	10 (30.3%)	14 (42.4%)	7 (21.2%)	0 (0.0%)	2 (6.1%)	9 (45.0%)	19 (95.0%)
BookKeeper 4.1.0	8	30 (3.8)	9 (29.0%)	17 (54.8%)	0 (0.0%)	2 (6.5%)	3 (9.7%)	6 (75.0%)	8 (100%)
Derby 10.7.1.1	10	18 (1.8)	7 (38.9%)	3 (16.7%)	0 (0.0%)	3 (16.7%)	5 (27.8%)	7 (70.0%)	9 (90.0%)
Derby 10.9.1.0	26	49 (1.9)	18 (36.7%)	12 (24.5%)	7 (14.3%)	9 (18.4%)	3 (6.1%)	17 (65.4%)	24 (92.3%)
Hibernate 3.5.0b2	3	7 (2.3)	3 (42.9%)	1 (14.3%)	3 (42.9%)	0 (0.0%)	0 (0.0%)	2 (66.7%)	3 (100%)
JabRef 2.6	3	4 (1.3)	3 (75.0%)	1 (25.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	3 (100.0%)	3 (100%)
jEdit 4.3	8	9 (1.1)	7 (77.8%)	1 (11.1%)	1 (11.1%)	0 (0.0%)	0 (0.0%)	6 (75.0%)	8 (100%)
Lucene 4.0	5	11 (2.2)	8 (72.7%)	3 (27.3%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	5 (100.0%)	5 (100%)
Mahout 0.8	7	11 (1.6)	5 (45.5%)	6 (54.5%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	5 (71.4%)	7 (100%)
muCommander 0.8.5	4	6 (1.5)	2 (33.3%)	4 (66.7%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	2 (50.0%)	4 (100%)
OpenJPA 2.0.1	6	8 (1.3)	5 (62.5%)	1 (12.5%)	1 (12.5%)	0 (0.0%)	1 (12.5%)	5 (83.3%)	6 (100%)
OpenJPA 2.2.0	4	12 (3.0)	2 (16.7%)	4 (33.3%)	5 (41.7%)	1 (8.3%)	0 (0.0%)	2 (50.0%)	4 (100%)
Pig 0.8.0	17	26 (2.2)	6 (23.1%)	5 (19.2%)	3 (11.5%)	0 (0.0%)	12 (46.2%)	5 (41.7%)	9 (75.0%)
Pig 0.11.1	12	46 (2.7)	8 (17.4%)	18 (39.1%)	5 (10.9%)	10 (21.7%)	5 (10.9%)	8 (47.1%)	16 (94.1%)
Solr 4.4.0	3	5 (1.7)	3 (60.0%)	2 (40.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	3 (100.0%)	3 (100%)
Tika 1.3	3	3 (1.0)	2 (66.7%)	1 (33.3%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	2 (66.7%)	3 (100%)
ZooKeeper 3.4.5	16	36 (2.3)	16 (44.4%)	16 (44.4%)	4 (11.1%)	0 (0.0%)	0 (0.0%)	13 (81.3%)	16 (100%)
All	*155*	*314 (2.0)*	*114 (36.2%)*	*109 (34.6%)*	*36 (11.4%)*	*25 (7.9%)*	*31 (9.8%)*	*100 (64.5%)*	147 (94.8%)

[a] In parenthesis, average of patched classes per bug report

[b] In parenthesis, percentage values

IV. RESULTS AND DISCUSSION

Before presenting and discussing the results, we explore our data set in order to understand the relationship between the stack traces and the patched classes of the bug reports. The data is summarized in Table II. In total, the 155 bug reports in our dataset are related to 314 patched classes (average=2.0 and median=1.0). Specifically, 81 bug reports (52.2%) are associated to one patched class, 38 (24.5%) to two patched classes, 35 (22.5%) to three to nine classes, and only one bug report points to ten patched classes.

From the patched classes, 36.2% (114 out of 314) are included in the stack traces of their respective bug reports, i.e., their distance with the stack trace (as described in Section II.B) is equal to zero (column four of Table II). These classes correspond to 100 of the bug reports, i.e., 64.5% of the bugs in our data set can be fixed in a class mentioned in the submitted stack trace (column nine of Table II). This is consistent with results reported in previous research [24]. From the rest of the patched classes, 34.6% (109 out of 314) are at distance one from the stack traces of their respective bug reports, 11.4% (36) are at distance two, and 7.9% (25) are at distance three to seven. Only 9.8% (31) of the patched classes are unreachable (in the call graph) from the classes included in stack traces submitted in the bug reports (i.e., their distance to the stack trace is equal to infinite). The 283 patched classes that are related to the stack traces correspond to 147 of the bug reports (see last column of Table II). This means that 94.8% of the bugs in our data set can be fixed in a class structurally related to the stack trace provided in the bug report, which confirms our initial assumption.

A. The Impact of λ and α on Lobster's Performance

In order to understand the effect of λ and α on Lobster's performance, we use different values for these parameters. Remember that λ defines the maximum considered distance when computing the structural similarity between a stack trace and a class, whereas α defines the weight of this structural similarity. Considering the distance values obtained in the exploration of the data set, we compare the performance of Lobster when using $\lambda \in \{1,2,3\}$ at $\alpha \in [0.1,1]$.

We use MRR and MAP for assessing the overall impact of the parameters in Lobster. Fig. 1 shows the trends of these measures at different values of λ and α when using all the queries. For every λ, the MRR and MAP values increase as α increases (except for $\alpha = 1$). According to Friedman's test, the MRR values for $\alpha \in [0.1,0.9]$ do not significantly differ between the sets of values $\lambda \in \{1,2,3\}$. The same situation occurs for the MAP values. When $\alpha = 1$, however, the MRR and MAP values are lower for all λ values.

We also analyze the effect of the parameters in the effectiveness. Table V reports Lobster's average and median effectiveness for different λ and α values. Remember that the lower the effectiveness is, the more effective the retrieval approach is. The lowest (i.e., best) and highest (i.e., worst) effectiveness for $\alpha \in (0,1]$ are obtained with $\lambda = 2$ and $\lambda = 3$, respectively.

When $\alpha = 1$, the effectiveness decreases (i.e., improves) as

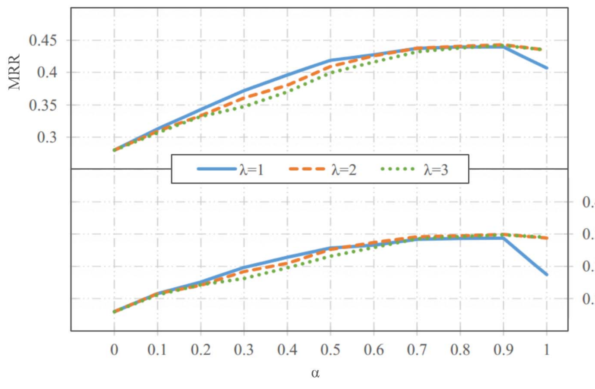

Fig. 1. MRR and MAP values obtained by Lobster on the entire data set with $\lambda \in \{1,2,3\}$ at different values of α.

TABLE III. MEAN RECIPROCAL RANK (MRR) OF LOBSTER$_{\lambda=1}$ FOR DIFFERENT VALUES OF α (THE ARROWS AND COLORS INDICATE INCREASE (↗), DECREASE (↘) AND NO CHANGE (→) IN THE MRR TREND WITH RESPECT TO THE PREVIOUS α)

System	Lucene (α=0)	α=0.1	α=0.2	α=0.3	α=0.4	α=0.5	α=0.6	α=0.7	α=0.8	α=0.9	sim$_{struct}$ (α=1)
ArgoUML 0.22	0.15	→ 0.15	↗ 0.18	↗ 0.20	↗ 0.21	↗ 0.25	→ 0.25	→ 0.25	→ 0.25	→ 0.25	↘ 0.24
BookKeeper 4.1.0	0.43	↗ 0.53	↗ 0.68	→ 0.68	→ 0.68	→ 0.68	→ 0.68	→ 0.68	→ 0.68	→ 0.68	↘ 0.67
Derby 10.7.1.1	0.22	↗ 0.23	↗ 0.27	↗ 0.34	↗ 0.36	↗ 0.38	→ 0.38	→ 0.38	→ 0.38	→ 0.38	↘ 0.37
Derby 10.9.1.0	0.11	↗ 0.14	↗ 0.17	↗ 0.20	↗ 0.30	↗ 0.32	↗ 0.34	↗ 0.36	→ 0.36	→ 0.36	↘ 0.35
Hibernate 3.5.0b2	0.36	↗ 0.39	↗ 0.67	→ 0.67	→ 0.67	→ 0.67	→ 0.67	→ 0.67	→ 0.67	→ 0.67	→ 0.67
JabRef 2.6	0.55	↗ 0.72	↗ 0.83	↘ 0.67	→ 0.67	→ 0.67	→ 0.67	→ 0.67	→ 0.67	→ 0.67	→ 0.67
jEdit 4.3	0.08	↗ 0.09	↘ 0.08	↗ 0.10	↗ 0.12	↗ 0.13	→ 0.13	→ 0.13	→ 0.13	→ 0.13	↘ 0.09
Lucene 4.0	0.45	↗ 0.46	→ 0.46	↗ 0.47	↗ 0.48	→ 0.48	→ 0.48	↗ 0.50	↗ 0.53	→ 0.53	→ 0.53
Mahout 0.8	0.32	↗ 0.33	↘ 0.31	↗ 0.37	↘ 0.32	↗ 0.35	↗ 0.45	→ 0.45	→ 0.45	→ 0.45	↘ 0.42
muCommander 0.8.5	0.76	→ 0.76	→ 0.76	→ 0.76	→ 0.76	→ 0.76	→ 0.76	→ 0.76	→ 0.76	→ 0.76	↘ 0.50
OpenJPA 2.0.1	0.39	↗ 0.45	↗ 0.48	→ 0.48	↗ 0.50	→ 0.50	→ 0.50	→ 0.50	→ 0.50	→ 0.50	→ 0.50
OpenJPA 2.2.0	0.05	↗ 0.08	↗ 0.11	↗ 0.31	→ 0.31	→ 0.31	↗ 0.33	↗ 0.39	→ 0.39	→ 0.39	↘ 0.38
Pig 0.8.0	0.29	↗ 0.34	→ 0.34	↗ 0.38	↗ 0.39	→ 0.39	→ 0.39	→ 0.39	→ 0.39	→ 0.39	↘ 0.29
Pig 0.11.1	0.17	↗ 0.18	→ 0.18	↗ 0.21	↗ 0.23	↗ 0.24	↗ 0.26	↗ 0.30	→ 0.30	→ 0.30	↘ 0.27
Solr 4.4.0	0.34	↗ 0.36	↗ 0.37	↗ 0.38	↗ 0.44	↗ 0.46	↗ 0.47	→ 0.47	→ 0.47	→ 0.47	→ 0.47
Tika 1.3	0.54	↗ 0.71	↗ 0.72	→ 0.72	↗ 0.73	→ 0.73	↘ 0.57	→ 0.57	→ 0.57	→ 0.57	↘ 0.40
ZooKeeper 3.4.5	0.57	↗ 0.64	↗ 0.67	↗ 0.70	↗ 0.71	↗ 0.81	→ 0.81	→ 0.81	↗ 0.82	→ 0.82	↘ 0.78
All	*0.28*	*↗ 0.31*	*↗ 0.34*	*↗ 0.37*	*↗ 0.40*	*↗ 0.42*	*↗ 0.43*	*↗ 0.44*	*→ 0.44*	*→ 0.44*	*↘ 0.41*

λ increases. In this case, we notice that λ acts as a smoothing factor that allows expanding the set of classes relevant to the bug report. This result indicates that the textual and the structural similarities capture complementary information. In addition, for the queries whose effectiveness is improved, the improvement is, in average, by 65.4 positions when $\lambda = 1$, 73.5 when $\lambda = 2$, and 71.2 when $\lambda = 3$, whereas for the queries whose effectiveness is degraded, it is, in average, by 2.52 positions when $\lambda = 1$, 73.0 and 71.2 when $\lambda = 3$.

We take a closer look at the MRR and MAP values of Lobster$_{\lambda=1}$ (i.e., Lobster with $\lambda = 1$), as it shows the most variance as α increases. Table III and Table IV show the MRR and MAP trends for each individual system. The trends are similar across all systems and consistent with the trends shown in Fig. 1, with some small differences. For example, the MRR for Mahout (which has seven queries) increases to 0.33 at $\alpha = 0.1$ due to an improvement (by 2 positions) in the effectiveness of the query q_6. At $\alpha = 0.2$, however, the MRR decreases to 0.31. This decrease is caused by the queries q_3 and q_7, whose effectiveness was degraded (by 2 and 4 positions, respectively), while the effectiveness of query q_6 was improved again (by 2 positions). At $\alpha = 0.3$, the MRR increases once more, but this time to 0.37. This is because of the effectiveness improvement of the queries q_6 and q_7 (by 4 and 3 positions, respectively). Note that q_7 is one of the queries whose effectiveness degraded before. We studied this query to understand the phenomenon. This query has two patched classes: `MapBackedArffModel`, included in the stack trace, and `ArffVectorIterable`, at distance one from it. `ArffVectorIterable` has a higher textual similarity to the query q_7 (i.e., 0.6) than `MapBackedArffModel` (i.e., 0.3), so for low values of α, it is ranked higher. However, as α increases `MapBackedArffModel`'s rank increases, since its structural similarity with the stack trace (i.e., 1) is higher than for `ArffVectorIterable` (i.e., 0), which explains the variations in the effectiveness of the query q_7. Going back to the MRR in Mahout, it decreases again at $\alpha = 0.4$ due to the effectiveness degradation of the query q_2 (by 4 positions). At $\alpha = 0.5$ and $\alpha = 0.6$, the MRR increases to 0.35 and 0.45,

TABLE IV. MEAN AVERAGE PRECISION (MAP) OF LOBSTER$_{\lambda=1}$ FOR DIFFERENT VALUES OF α (THE ARROWS AND COLORS INDICATE INCREASE (↗), DECREASE (↘) AND NO CHANGE (→) IN THE MAP TREND WITH RESPECT TO THE PREVIOUS α)

System	Lucene (α=0)	α=0.1	α=0.2	α=0.3	α=0.4	α=0.5	α=0.6	α=0.7	α=0.8	α=0.9	sim$_{struct}$ (α=1)
ArgoUML 0.22	0.08	↗ 0.09	↗ 0.10	↗ 0.13	→ 0.13	↗ 0.16	↗ 0.17	→ 0.17	→ 0.17	→ 0.17	↘ 0.14
BookKeeper 4.1.0	0.34	↗ 0.38	↗ 0.43	↗ 0.44	→ 0.44	→ 0.44	→ 0.44	→ 0.44	→ 0.44	→ 0.44	↘ 0.26
Derby 10.7.1.1	0.16	↗ 0.17	↗ 0.19	↗ 0.22	↗ 0.23	↗ 0.24	→ 0.24	→ 0.24	→ 0.24	→ 0.24	↘ 0.23
Derby 10.9.1.0	0.10	↗ 0.11	↗ 0.14	↗ 0.18	↗ 0.25	↗ 0.26	↗ 0.29	↗ 0.31	→ 0.31	→ 0.31	↘ 0.29
Hibernate 3.5.0b2	0.35	↗ 0.36	↗ 0.45	→ 0.45	→ 0.45	→ 0.45	→ 0.45	→ 0.45	→ 0.45	→ 0.45	↘ 0.44
JabRef 2.6	0.49	↗ 0.67	↗ 0.78	↘ 0.69	→ 0.69	→ 0.69	→ 0.69	→ 0.69	→ 0.69	→ 0.69	↘ 0.58
jEdit 4.3	0.08	→ 0.08	→ 0.08	↗ 0.10	↗ 0.12	↗ 0.14	→ 0.14	→ 0.14	→ 0.14	→ 0.14	↘ 0.09
Lucene 4.0	0.44	→ 0.44	↗ 0.45	→ 0.45	↗ 0.46	→ 0.46	→ 0.46	↗ 0.47	↗ 0.48	↗ 0.49	↘ 0.48
Mahout 0.8	0.30	↗ 0.31	→ 0.31	↗ 0.36	↘ 0.31	↗ 0.32	↗ 0.35	→ 0.35	→ 0.35	→ 0.35	↘ 0.30
muCommander 0.8.5	0.65	→ 0.65	→ 0.65	→ 0.65	→ 0.65	→ 0.65	→ 0.65	→ 0.65	→ 0.65	→ 0.65	↘ 0.38
OpenJPA 2.0.1	0.37	↗ 0.40	↗ 0.43	→ 0.43	↗ 0.45	→ 0.45	→ 0.45	→ 0.45	→ 0.45	→ 0.45	↘ 0.44
OpenJPA 2.2.0	0.04	↗ 0.07	↗ 0.10	↗ 0.30	→ 0.30	→ 0.30	↗ 0.33	↗ 0.39	→ 0.39	→ 0.39	↘ 0.38
Pig 0.8.0	0.21	↗ 0.23	↗ 0.24	↗ 0.25	→ 0.25	↗ 0.26	→ 0.26	→ 0.26	→ 0.26	→ 0.26	↘ 0.18
Pig 0.11.1	0.14	→ 0.14	→ 0.14	→ 0.14	↗ 0.16	→ 0.16	↗ 0.18	↗ 0.21	→ 0.21	→ 0.21	↘ 0.15
Solr 4.4.0	0.34	↗ 0.35	↗ 0.36	→ 0.36	↗ 0.39	↗ 0.40	↗ 0.41	→ 0.41	→ 0.41	→ 0.41	↘ 0.40
Tika 1.3	0.54	↗ 0.71	↗ 0.72	→ 0.72	↗ 0.73	→ 0.73	↘ 0.57	→ 0.57	→ 0.57	→ 0.57	↘ 0.40
ZooKeeper 3.4.5	0.48	↗ 0.55	↗ 0.56	↗ 0.57	↗ 0.58	↗ 0.62	→ 0.62	→ 0.62	↗ 0.63	→ 0.63	↘ 0.54
All	*0.23*	*↗ 0.26*	*↗ 0.28*	*↗ 0.30*	*↗ 0.31*	*↗ 0.33*	*→ 0.33*	*↗ 0.34*	*→ 0.34*	*→ 0.34*	*↘ 0.29*

TABLE V. AVERAGE EFFECTIVENESS OF LOBSTER FOR DIFFERENT VALUES OF λ AND α, AND NUMBER OF BUG REPORTS WHERE LOBSTER IS BETTER THAN, SAME AS AND WORSE THAN LUCENE

α	λ=1				λ=2				λ=3			
	Avg. Effect. [a]	***Better*** [b]	***Same*** [b]	***Worse*** [b]	***Avg. Effect.*** [a]	***Better*** [b]	***Same*** [b]	***Worse*** [b]	***Avg. Effect.*** [a]	***Better*** [b]	***Same*** [b]	***Worse*** [b]
0	99.9 (16)	-	-	-	99.9 (16)	-	-	-	99.9 (16)	-	-	-
0.1	82.5 (12)	54 (34.8%)	83 (53.5%)	18 (11.6%)	79.0 (12)	78 (50.3%)	55 (35.5%)	22 (14.2%)	83.1 (12)	83 (53.5%)	54 (34.8%)	18 (11.6%)
0.2	76.1 (8)	61 (39.4%)	63 (40.6%)	31 (20.0%)	69.2 (10)	84 (54.2%)	50 (32.3%)	21 (13.5%)	75.2 (11)	88 (56.8%)	47 (30.3%)	20 (12.9%)
0.3	72.6 (6)	65 (41.9%)	58 (37.4%)	32 (20.6%)	64.7 (8)	88 (56.8%)	41 (26.5%)	26 (16.8%)	72.7 (9)	91 (58.7%)	43 (27.7%)	21 (13.5%)
0.4	71.2 (5)	67 (43.2%)	55 (35.5%)	33 (21.3%)	62.5 (6)	91 (58.7%)	38 (24.5%)	26 (16.8%)	71.4 (8)	91 (58.7%)	40 (25.8%)	24 (15.5%)
0.5	70.7 (5)	68 (43.9%)	54 (34.8%)	33 (21.3%)	63.2 (5)	91 (58.7%)	38 (24.5%)	26 (16.8%)	71.1 (6)	92 (59.4%)	38 (24.5%)	25 (16.1%)
0.6	70.5 (4)	68 (43.9%)	53 (34.2%)	34 (21.9%)	63.7 (5)	91 (58.7%)	36 (23.2%)	28 (18.1%)	73.0 (5)	92 (59.4%)	39 (25.2%)	24 (15.5%)
0.7	70.4 (4)	68 (43.9%)	53 (34.2%)	34 (21.9%)	63.3 (4)	91 (58.7%)	36 (23.2%)	28 (18.1%)	73.5 (4)	92 (59.4%)	36 (23.2%)	27 (17.4%)
0.8	70.4 (4)	68 (43.9%)	53 (34.2%)	34 (21.9%)	63.1 (4)	91 (58.7%)	35 (22.6%)	29 (18.7%)	73.3 (4)	92 (59.4%)	36 (23.2%)	27 (17.4%)
0.9	70.4 (4)	68 (43.9%)	53 (34.2%)	34 (21.9%)	63.1 (4)	91 (58.7%)	35 (22.6%)	29 (18.7%)	73.2 (4)	92 (59.4%)	35 (22.6%)	28 (18.1%)
1	813.7 (5)	68 (43.9%)	28 (18.1%)	59 (38.1%)	374.2 (4)	91 (58.7%)	31 (20.0%)	33 (21.3%)	263.3 (4)	92 (59.4%)	31 (20.0%)	32 (20.6%)

[a] In parenthesis, median values
[b] In parenthesis, percentage values

respectively, since the effectiveness of the query q_3 improves (by 2 positions each time). For the subsequent values of α this MRR holds until decreasing to 0.42 at $\alpha = 1$, where the patched classes of the queries q_2 and q_3 are not retrieved, since they are not included in the respective stack traces (remember that $\lambda = 1$).

Note that the maximum MRR and MAP values are not always reached at the same α value in all subject systems, yet it is above $\alpha = 0.2$ in all cases. Across the systems, the highest MRR and MAP values can be found at $\alpha = 0.7$ and are maintained until $\alpha = 0.9$.

In conclusion, we recommend using values between [0.7, 0.9] for α, and 2 for λ. Study replications are needed to confirm these findings and future work will define data-driven heuristics for determining the best values for α and λ.

B. Lobster vs. Classic TR-based Bug Localization

It is common in previous research on static techniques [10] to use plain TR- based ranking as the baseline of comparison. In order to determine whether Lobster improves TR-based bug localization, we use Lucene (i.e., Lobster with $\alpha = 0$) as the baseline. The analysis from the previous section shows that the MRR and MAP improve for all Lobster configurations, compared to Lucene. We compare now Lobster and Lucene in terms of effectiveness, since this measure allows for a finer granularity analysis than MRR or MAP.

We use all λ and α values to compare Lobster's and Lucene's effectiveness, rather than selecting only the best configuration. The average effectiveness obtained by Lucene is 99.9 (median=16). By comparison (see Table V), Lobster outperforms Lucene when using $\alpha \in (0,1)$ and $\lambda = \{1,2,3\}$, by lowering the average effectiveness rank by 16.9 to 37.4 positions (depending on the λ and α values). Lobster also improves the median effectiveness, by lowering it from 16 (obtained by Lucene) to four (4) in the best case (e.g., for $\alpha \in (0.6, 0.9]$) and 12 in the worst case (i.e., with $\alpha = 0.1$). In practical terms, a median effectiveness of four (4) means that for 50% of the bugs, the class to be changed is retrieved among the top four documents. We used the Wilcoxon test to evaluate the difference between the effectiveness obtained by Lucene and each Lobster$_{\lambda,\alpha}$ with $\lambda \in \{1,2,3\}$ and $\alpha \in (0,1)$. We found that the difference is statistically significant in each case when $\alpha > 0.2$ (p-value<0.05). The average magnitude of such difference is small, according to the Cliff's delta (d=–0.23). The small magnitude is not a concern, as Lucene's median results are already good in many cases.

It is common for text retrieval techniques, in general, and TR-based bug localization techniques, in particular, to achieve improvements for a subset of queries from a benchmark, while performing worse than a baseline on another subset of queries. Therefore we also analyze the results in terms of such subsets of queries, not just in terms of average improvement.

The number of queries where Lobster (with different values of λ and α) performs better than, the same as, or worse than Lucene is also reported in Table V and summarized in Fig. 2. Lobster improves, in average, Lucene's effectiveness obtained for 52.6% of the queries and maintains it for 29.0% of them. Only in 18.4% of the cases, in average, Lobster's effectiveness degrades compared to Lucene's. Specifically, the largest subset of queries with effectiveness improvement is achieved

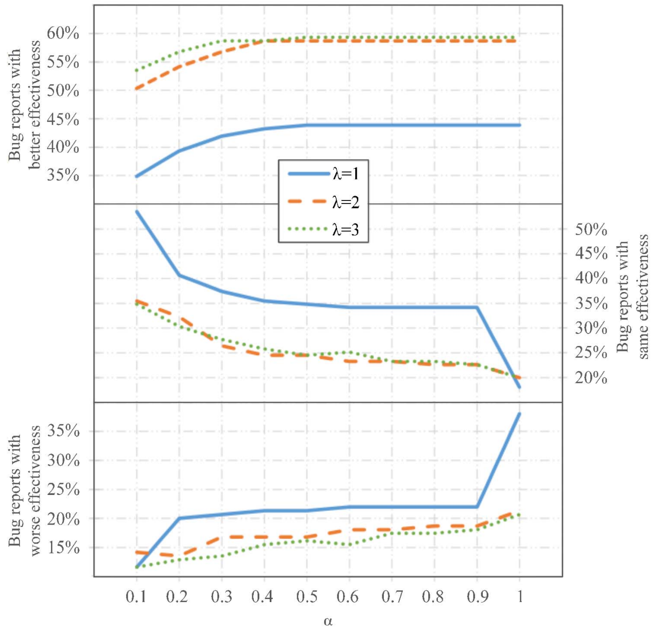

Fig. 2. Percentage of bug reports where Lobster obtains better, the same and worse effectiveness than Lucene, with $\lambda \in \{1,2,3\}$ at different values of α

with $\lambda = 3$, where Lobster improves the effectiveness for 59.4% (92 out of 155) of the queries (at $\alpha \in [0.5,1.0]$) by 81.0 positions, in average (median=28). For each subset of improved queries, there is a subset of queries with lower performance than the baseline. In this case (i.e., $\lambda = 3$ and $\alpha \in [0.5,1.0]$), these subsets contain between 15.5% (24) and 20.6% (32) of the queries, whose average effectiveness deteriorates by 274.1 positions, in average (median=57). Interestingly, these cases do not correspond to the highest improvement in average (and median) effectiveness, which is achieved when $\lambda = 2$ and $\alpha \in [0.8,0.9]$—that is, 63.1 average effectiveness (median=4). In such cases, the effectiveness improves by 80.1 positions, in average (median=28), for 58.7% (91) of the queries, and degrades by 60.1 positions, in average (median=24), for 18.7% (29) of the queries. According to the Cliff's delta, the magnitude of the effectiveness difference is small for these 18.7% of queries (*d*=0.20), whereas for the 58.7% of the queries for which the effectiveness was improved, the difference is large (*d* =–0.45).

In contrast, the smallest subset with effectiveness improvement is obtained with $\lambda = 1$ and $\alpha = 0.1$, where Lobster improves the effectiveness for 34.8% (54) queries, by lowering it 50.48 positions, in average (median=12). At the same time, it degrades the effectiveness for 11.6% (18) of them, by raising it only 1.61 positions, in average (median=1). According to the Cliff's delta, the magnitude of the effectiveness difference is negligible for these 12% of queries (*d*=0.80), whereas for the 35% of the queries for which the effectiveness is improved, the difference is small (*d* =–0.27). This is not unexpected, as at $\alpha = 0.1$, the textual similarity in Lobster's similarity measure has the highest weight. Hence the structural similarity brings little noise (i.e., boosting false positives) into the total similarity, which explains the small subsets where the effectiveness degrades.

As observed in the MRR and MAP analysis, Lobster$_{\lambda=1}$ improves most with respect to Lucene for most systems, when using $\alpha \in [0.7,0.9]$. In such cases, the average and median effectiveness are improved with a reduction of 29.5 and 12 positions, respectively. In particular, the performance is improved by Lobster$_{\lambda=1}$ for 43.9% (68 out of 155) of the bug reports, where the effectiveness is lowered 68.56 positions in average (median=19.5), and maintained for 34.2% (53). For 21.9% (34) of the bug reports, however, Lobster$_{\lambda=1}$ degrades the effectiveness by 2.76 positions, in average (median=2). The magnitude of the effectiveness difference is negligible for these 21.9% of queries (*d*=0.10) according to the Cliff's delta, whereas for the 43.9% of the queries for which the effectiveness was improved, the difference is large (*d* =–0.73). Note that this results are better than the ones obtained by Lobster$_{\lambda=1}$ with $\alpha = 0.1$. In this sense, using $\alpha \in [0.7,0.9]$ provides the most gain in effectiveness for Lobster$_{\lambda=1}$. This result also supports our analysis and conclusion from the previous section, based on MRR and MAP data.

We analyze in more details Lobtser's performance, for each system. We selected Lobster with $\lambda = 1$ for further investigation, since it provides the smallest overall improvement with respect to Lucene, i.e., this is a worse-case scenario for Lobster. When comparing Lobster$_{\lambda=1}$ and Lucene in terms of effectiveness (Table VI), we observe that Lobster$_{\lambda=1}$ outperforms Lucene by lowering the average and median effectiveness for all $\alpha \in (0,1)$ in each system, except in muCommander, where the average effectiveness increases by 0.2 with Lobster $_{\lambda=1}$ for every $\alpha \in (0,1)$. The general improvement in the effectiveness is caused by the boost in the rank of the patched classes that are included in the stack traces of their respective bug reports.

When $\alpha = 0.1$, Lobster$_{\lambda=1}$ improves or maintains Lucene's effectiveness for all the queries of ten of the subject systems (i.e., Hibernate, JabRef, jEdit, Lucene, Mahout, both versions of OpenJPA, Solt, Tika, and ZooKeeper). Only for one of the systems (i.e., muCommander), there was no effectiveness improvement, and actually one of its queries got worse results than with Lucene. When examining the results of this query, we found that the difference in the effectiveness was only of one position (36 with Lucene and 37 with Lobster$_{\lambda=1}$), and due to the fact that the only patched class of this query is not in the

TABLE VI. AVERAGE EFFECTIVENESS OF LOBSTER $_{\lambda=1}$ FOR DIFFERENT VALUES OF α (IN PARENTHESIS, THE MEDIAN VALUES)

System	**Lucene (α=0)**	**α=0.1**	**α=0.2**	**α=0.3**	**α=0.4**	**α=0.5**	**α=0.6**	**α=0.7**	**α=0.8**	**α=0.9**	**sim$_{struct}$ (α=1)**
ArgoUML 0.22	168 (78)	141.9 (32.5)	127 (29.5)	123.9 (15.5)	123.5 (12.5)	123 (12.5)	122.9 (12.5)	122.9 (12.5)	122.9 (12.5)	122.9 (12.5)	900.6 (1635)
BookKeeper 4.1.0	7.6 (3)	6.9 (2.5)	6.5 (1)	6.5 (1)	6.5 (1)	6.5 (1)	6.5 (1)	6.5 (1)	6.5 (1)	6.5 (1)	147.8 (1)
Derby 10.7.1.1	154.3 (55.5)	123.9 (29.5)	109.2 (26.5)	100.8 (17.5)	97.5 (6)	96.7 (5)	96.7 (5)	96.7 (5)	96.7 (5)	96.7 (5)	914.9 (5)
Derby 10.9.1.0	203 (57)	193.5 (28.5)	189.3 (18)	181.8 (13)	179.1 (6.5)	178.5 (6)	178.1 (5.5)	178 (5.5)	177.9 (5.5)	177.9 (5.5)	1,087 (6)
Hibernate 3.5.0b2	80 (12)	78 (6)	76.3 (1)	76.3 (1)	76.3 (1)	76.3 (1)	76.3 (1)	76.3 (1)	76.3 (1)	76.3 (1)	1,436.3 (1)
JabRef 2.6	3.3 (2)	2.7 (1)	1.3 (1)	1.7 (2)	1.7 (2)	1.7 (2)	1.7 (2)	1.7 (2)	1.7 (2)	1.7 (2)	1.7 (2)
jEdit 4.3	83.5 (46)	46.5 (35.5)	31.3 (25.5)	20.8 (17)	15.9 (11)	14 (10.5)	14 (10.5)	13.9 (10)	13.9 (10)	13.9 (10)	260.9 (12)
Lucene 4.0	212.2 (5)	15.4 (5)	10.2 (5)	8.6 (4)	7.4 (4)	7 (4)	6.8 (4)	6.6 (3)	6 (2)	6 (2)	6 (2)
Mahout 0.8	16.3 (4)	16 (4)	16.6 (6)	15.6 (4)	16.1 (5)	15.6 (4)	15.3 (4)	15.3 (4)	15.3 (4)	15.3 (4)	933.3 (4)
muCommander 0.8.5	9.8 (1)	10 (1)	10 (1)	10 (1)	10 (1)	10 (1)	10 (1)	10 (1)	10 (1)	10 (1)	722 (722)
OpenJPA 2.0.1	23.7 (10.5)	21.7 (6.5)	19.8 (3)	19.7 (3)	19.3 (2.5)	19.3 (2.5)	19.3 (2.5)	19.3 (2.5)	19.3 (2.5)	19.3 (2.5)	741.8 (2.5)
OpenJPA 2.2.0	55.8 (31)	46.8 (14)	43.8 (9)	42.8 (9)	44 (11.5)	44.3 (12)	43.8 (11)	43.3 (10)	43.3 (10)	43.3 (10)	2,478.3 (2,478.5)
Pig 0.8.0	55.9 (23)	48.1 (18.5)	47.3 (12.5)	46.7 (8.5)	46.4 (7.5)	46.3 (7)	46.3 (7)	46.3 (7)	46.3 (7)	46.3 (7)	1,462.8 (2,506)
Pig 0.11.1	104.9 (40)	101.5 (30)	89.8 (25)	86 (15)	83.8 (11)	82.4 (6)	82 (6)	81.9 (6)	81.9 (6)	81.9 (6)	1,110.5 (2,095)
Solr 4.4.0	72 (46)	36.7 (18)	18.7 (11)	11 (9)	5 (5)	4 (4)	3.7 (4)	3.7 (4)	3.7 (4)	3.7 (4)	3.7 (4)
Tika 1.3	4 (2)	3 (1)	2.7 (1)	2.7 (1)	2.3 (1)	2.3 (1)	2.7 (2)	2.7 (2)	2.7 (2)	2.7 (2)	196 (5)
ZooKeeper 3.4.5	4.2 (2.5)	2.8 (1.5)	2.7 (1)	2.5 (1)	2.4 (1)	2.2 (1)	2.2 (1)	2.2 (1)	2.1 (1)	2.1 (1)	141.9 (1)
All	*99.9 (16)*	*82.5 (12)*	*76.1 (8)*	*72.6 (6)*	*71.2 (5)*	*70.7 (5)*	*70.5 (4)*	*70.4 (4)*	*70.4 (4)*	*70.4 (4)*	*813.7 (5)*

stack trace, but at distance one from it (remember that $\lambda = 1$, which is a worst case scenario for Lobster). For the other six systems (i.e., ArgoUML, BookKeeper, and all versions of Derby and Pig) the results are mixed, with some queries being improved by Lobster, while others not. This is expected, as not all the patched classes are included in the stack traces.

The worst scenario of Lobster$_{\lambda=1}$ is given by $\alpha = 1$, i.e., using only the structural similarity, where we found a significant deterioration in the average effectiveness (813.7 for Lobster vs. 99.9 for Lucene). We noticed that Lucene achieves better results in 38.1% (59 out of 155) of the queries. This happens when the parts of code that need to be fixed are not structurally related to the stack traces submitted in the bug reports. According to the Cliff's delta, the magnitude of the effectiveness difference in this case is large (d=0.86). Note, however, that the structural similarity improves the effectiveness for 43.9% (68) of the queries. In this case, the magnitude of the effectiveness difference is also large (d=–0.75). Once again, this fact indicates that Lucene (or, for that matter, any other TR approach) and the structural similarity capture different types of information. The structural similarity is meant to be used as a *complement* to the textual similarity, and our result underlines this paradigm. This is confirmed by the results per system, where we observed that when $\alpha = 1$, Lucene's effectiveness is improved or maintained by Lobster$_{\lambda=1}$ for all the queries of only two of the systems (i.e., Lucene and Solr). Once again, only in muCommander there is no effectiveness improvement in any query, and this time two of them get worse results than with Lucene. Such queries correspond to bug reports whose patched classes are not included in the stack trace, and therefore they are no retrieved. These classes are immediate neighbors of the classes in the stack traces, which means that the results for these queries improve by using Lobster with a greater value of λ.

C. *Threats to Validity*

Several threats to validity may affect the results of our empirical evaluation. Threats to *construct validity* concern the relationship between theory and observation. We evaluated Lobster using different performance measures (i.e., effectiveness, MRR and MAP) widely used in concept and feature location research [10].

Threats to *conclusion validity* refer to the relationship between treatment and outcome. We made no assumption on the distribution of the data and used a non-parametric test for assessing the significance of the results, when appropriate.

Threats to the *internal validity* of our study concern factors that can influence our results. For measuring the performance of Lobster we used measures that depend on the rank of patched classes. We used two sets of data in our data set: benchmark data [10] heavily used to evaluate bug localization techniques, and bug reports automatically extracted from the online tracking system of the subject systems In both cases, some patched classes could be missing from the oracle of each system. Regarding our implementation choices, we used a call graph extracted from the binary files of the subject systems. We expect little impact on the results when using other call graph extractors, yet we do not expect them to degrade Lobster's performance.

Threats to the *external validity* refer to the generalization of our results. We used the bug reports from 17 versions of 14 software systems. These systems are from different domains and sizes. However, the number of bug reports for some of the systems is low. A larger set of bug reports would strengthen the results. Also, we only used a single TR engine (i.e., Lucene). The results might vary when using other TR engines.

V. RELATED WORK

Relevant to our approach is the research on concept and feature location. A comprehensive survey on this topic was recently published by Dit et al. [10]. Our focus is on the research that applies TR to support concept and feature location. Such approaches have been also recently surveyed [16]. Particularly related to our work is the research combining TR models with data extracted with other techniques, such as: static analysis (e.g., [1, 2, 6, 8, 18, 21, 23, 25, 28]), dynamic analysis (e.g., [11, 15, 19]), and data mining (e.g., [7, 27, 29]).

TR approaches applying static analysis techniques make use of structural information extracted from the code or binaries of a software system to boost the relevance of specific code elements to a query. The structural information statically extracted is diverse. For example, some TR techniques consider abstract syntax attributes [2, 21] to give higher priority to the terms in particular constructs (e.g., class and method names). Other TR techniques use, instead, control or data flow dependencies (e.g., [1, 6, 8, 18, 23, 25, 28]) to increase the relevance of code elements that have a dependency relationship with code elements that are textually relevant to a query. Similar to these approaches, Lobster combines TR and static analysis for bug localization: it takes into consideration the textual similarity between bug reports and code elements and it also makes use of a call graph built from static analysis on software systems. In a different way, Lobster also takes advantage of the stack traces submitted in bug reports to boost the relevance of the code elements that have a dependency relationship with the stack trace elements.

Given the use of stack traces, Lobster also relates to concept and feature location approaches combining TR and dynamic analysis techniques (e.g., [11, 15, 19]). These techniques rely on execution traces captured while exercising the desired (or undesired) features of the software. TR is used to retrieve code elements present in these traces, rather than in the entire software corpus. While these techniques have the best effectiveness, it is not always practical to capture execution traces in real time. Lobster, overcomes this problem by relying on stack traces already present in bug reports and static analysis of the existing code.

TR-based concept and feature location approaches relying on repository mining techniques make use of the information stored in issue trackers and control version systems to find patterns in the change history of software systems (e.g., [7, 27, 29]). For example, Zhou et al. [29] proposed a revised VSM-based model combined with a ranking based on similar bug reports to find code files relevant to a bug report. The assumption is that similar bugs are likely fixed in related files. One of the most related technique to Lobster is the one proposed by Davis and Roper [7], which considers the cosine similarity between bug descriptions and code methods, a bug language classifier, the number of previous bugs related to the method, and the inverse of the position of the method in the stack traces submitted in bug reports, in order to retrieve a ranked list of methods related to a bug report. Stack traces are also used by Wang et al. [27] to identify groups of correlated crash types, which in turn are used to identify buggy files. Unlike these approaches, Lobster does not require mining software repositories, since it only makes use of the information found in the bug report at hand. Moreover, Lobster finds parts of code that are related to the code elements listed in the stack traces submitted in bug reports. Like these approaches, Lobster only works when stack traces are present in the bug reports. In their absence, Lobster acts like a classic TR-based bug localization technique.

VI. Conclusions and Future Work

We empirically found that utilizing information from the stack traces present in bug reports improves TR-based bug localization. The technique that we implemented and evaluated, Lobster, finds code elements related to the ones present in the stack traces and promotes them during text-based retrieval of source code documents. The improvements are achieved with little overhead, as extracting the stack traces requires less effort than capturing execution traces or mining large repositories. Considering the results, we strongly advocate in favor of reporting stack traces with all bug reports.

The positive results are promising and warrant future work, which, we anticipate, will lead to further improvements and strengthening their validity. We plan to experiment with other ways to compute the structural similarity and combinations with textual similarities. In terms of evaluation, we plan to use larger data sets and additional baselines (e.g., [29] and [23]). Finally, we plan to develop data-driven heuristics for determining the best values for Lobster's parameters and to implement Lobster using other TR approaches.

Acknowledgement

This work was supported in part by grants from the National Science Foundation (CCF-1017263 and CCF-0845706).

References

[1] Ali, N., Sabane, A., Gueheneuc, Y.-G., and Antoniol, G., "Improving Bug Location Using Binary Class Relationships", in Proc. of 12th Intl. Working Conf. on Source Code Analysis and Manipulation (SCAM'12), 2012.

[2] Bassett, B. and Kraft, N. A., "Structural Information Based Term Weighting in Text Retrieval for Feature Location", in Proc. of 21st IEEE Intl. Conf. on Prog. Comprehension (ICPC'13), 2013, pp. 133-141.

[3] Bettenburg, N., Premraj, R., Zimmermann, T., and Kim, S., "Extracting Structural Information from Bug Reports", in Proc. of 5th Intl. Working Conf. on Mining Softw. Rep. (MSR'08), Leipzig, Germany, 2008.

[4] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E., "The Concept Assignment Problem in Program Understanding", in Proc. of 15th IEEE/ACM Intl. Conf. on Softw. Eng. (ICSE'93), Baltimore, MD, USA, 17-21 May 1993 1993, pp. 482-498.

[5] Blei, D. M., Ng, A. Y., and Jordan, M. I., "Latent Dirichlet Allocation", *Jnl. of Mach. Learn. Research*, vol. 3, March 2003 2003, pp. 993-1022.

[6] Chen, K. and Rajlich, V., "Case Study of Feature Location Using Dependence Graph", in Proc. of 8th IEEE Intl. Wksp. on Prog. Comprehension (IWPC'00), 2000, pp. 241-249.

[7] Davis, S. and Roper, M., "Bug localisation through diverse sources of information", in Proc. of IEEE Intl. Symp. on Softw. Reliab. Eng. Wksps. (ISSREW'13), Pasadena, CA, USA, 4-7 Nov 2013, pp. 123-131.

[8] Davis, S., Roper, M., and Wood, M., "Comparing text-based and dependency-based approaches for determining the origins of bugs", *Jnl. of Softw.: Evol. and Proc. (JSEP)*, vol. 26, no. 1, January 2014, pp. 107-139.

[9] Deerwester, S., Dumais, S., Furnas, G. W., Landauer, T., and Harshman, R., "Indexing by Latent Semantic Analysis", *Jnl. of the American Society for Information Science*, vol. 41, 1990, pp. 391-407.

[10] Dit, B., Revelle, M., Gethers, M., and Poshyvanyk, D., "Feature Location in Source code: A Taxonomy and Survey", *Jnl. of Softw.: Evol. and Proc. (JSEP)*, vol. 25, no. 1, 2012, pp. 53–95.

[11] Dit, B., Revelle, M., and Poshyvanyk, D., "Integrating Information Retrieval, Execution and Link Analysis Algorithms to Improve Feature Location in Software", *Emp. Softw. Eng. (EMSE)*, vol. 18, no. 2, 2013, pp. 277-309.

[12] Grissom, R. J. and Kim, J. J., *Effect Sizes for Research: Univariate and Multivariate Applications*, Taylor & Francis, 2005.

[13] Hatcher, E. and Gospodnetić, O., *Lucene in Action*, Manning Publications, 2004.

[14] Hill, E., Pollock, L., and Vijay-Shanker, K., "Investigating How to Effectively Combine Static Concern Location Techniques", in Proc. of 3rd IEEE/ACM Intl. Wksp on Search-driven Dev.: Users, Infrastructure, Tools, and Evaluation (SUITE'11), Honolulu, HI, 28 May 2011, pp. 37-40.

[15] Liu, D., Marcus, A., Poshyvanyk, D., and Rajlich, V., "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace", in Proc. of 22nd IEEE/ACM Intl. Conf. on Auto. Softw. Eng. (ASE'07), Atlanta, Georgia, USA, November 5-9 2007, pp. 234-243.

[16] Marcus, A. and Haiduc, S., "Text Retrieval Approaches for Concept Location in Source Code", in *Softw. Eng.: Lecture Notes in Computer Science*, 2013, pp. 126-158.

[17] Moreno, L., Bandara, W., Haiduc, S., and Marcus, A., "On the Vocabulary Relationship between Bug Reports and Source Code", in *29th IEEE Intl. Conf. on Software Maintenance (ICSM)*. Eindhoven, The Netherlands, 2013, pp. 425-455.

[18] Petrenko, M. and Rajlich, V., "Concept location using program dependencies and information retrieval (DepIR)", *Information and Software Technology*, vol. 55, no. 4, 2013, pp. 651-659.

[19] Poshyvanyk, D., Gueheneuc, Y. G., Marcus, A., Antoniol, G., and Rajlich, V., "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval", *IEEE Trans. On Softw. Eng. (TSE)*, vol. 33, no. 6, 2007, pp. 420-432.

[20] Rajlich, V. and Wilde, N., "The Role of Concepts in Program Comprehension", in Proc. of IEEE Intl. Wksp. on Prog. Comprehension (IWPC'02), 2002, pp. 271-278.

[21] Saha, R. K., Lease, M., Khurshid, S., and Peryy, D. E., "Improving bug localization using structured information retrieval", in Proc. of IEEE/ACM 28th Intl. Conf. on Auto. Softw. Eng. (ASE'13), Silicon Valley, CA, USA, 11-15 Nov 2013, pp. 345-355.

[22] Salton, G. and McGill, M., *Introduction to Modern Information Retrieval*, McGraw-Hill, 1983.

[23] Scanniello, G. and Marcus, A., "Clustering Support for Static Concept Location in Source Code", in Proc. of 19th IEEE Intl. Conf. on Progr. Comprehension (ICPC'11), Kingston, ON, June 22-24 2011, pp. 1-10.

[24] Schröter, A., Bettenburg, N., and Premraj, R., "Do stack traces help developers fix bugs?", in Proc. of 7th IEEE Working Conf. on Mining Soft. Rep. (MSR'10), Cape Town, South Africa, 2-3 May 2010, pp. 118-121.

[25] Shao, P. and Smith, R. K., "Feature Location by IR Modules and Call Graph", in Proc. of 47th ACM Annual Southeast Reg. Conf. (ACM-SE'09), Clemson, SC, 19-21 March 2009.

[26] Sheskin, D., *Handbook of parametric and nonparametric statistical procedures*, 4 ed., Chapman & Hall CRC 2007.

[27] Wang, S., Khomh, F., and Zou, Y., "Improving bug localization using correlations in crash reports", in Proc. of 10th Wrk. Conf. on Mining Softw. Rep. (MSR'10), San Franc., CA, USA, 18-26 May 2013, pp. 247-256.

[28] Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F., "SNIAFL: Towards a Static Non-interactive Approach to Feature Location", *ACM Trans. on Softw. Eng. and Method. (TOSEM)*, vol. 15, no. 2, 2006, pp. 195-226.

[29] Zhou, J., Zhang, H., and Lo, D., "Where Should the Bugs Be Fixed? More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports", in Proc. of 34th Intl. Conf. on Softw. Eng. (ICSE'12), Zurich, Switzerland, 2012, pp. 14-24.