Presented at 2020 IEEE/ACM 17th International Conference on Mining Software Repositories

A Large-Scale Comparative Evaluation of IR-Based Tools for Bug Localization

Shayan A. Akbar
Purdue University
West Lafayette, IN, USA
sakbar@purdue.edu

Avinash C. Kak
Purdue University
West Lafayette, IN, USA
kak@purdue.edu

ABSTRACT

This paper reports on a large-scale comparative evaluation of IR-based tools for automatic bug localization. We have divided the tools in our evaluation into the following three generations: (1) The first-generation tools, now over a decade old, that are based purely on the Bag-of-Words (BoW) modeling of software libraries. (2) The somewhat more recent second-generation tools that augment BoW-based modeling with two additional pieces of information: historical data, such as change history, and structured information such as class names, method names, etc. And, finally, (3) The third-generation tools that are currently the focus of much research and that also exploit proximity, order, and semantic relationships between the terms. It is important to realize that the original authors of all these three generations of tools have mostly tested them on relatively small-sized datasets that typically consisted no more than a few thousand bug reports. Additionally, those evaluations only involved Java code libraries. The goal of the present paper is to present a comprehensive large-scale evaluation of all three generations of bug-localization tools with code libraries in multiple languages. Our study involves over 20,000 bug reports drawn from a diverse collection of Java, C/C++, and Python projects. Our results show that the third-generation tools are significantly superior to the older tools. We also show that the word embeddings generated using code files written in one language are effective for retrieval from code libraries in other languages.

KEYWORDS

source code search, word embeddings, information retrieval, bug localization

ACM Reference Format:
Shayan A. Akbar and Avinash C. Kak. 2020. A Large-Scale Comparative Evaluation of IR-Based Tools for Bug Localization. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3379597.3387474

1 INTRODUCTION

Retrieving relevant source code files from software libraries in response to a bug report query plays an important role in the maintenance of a software project. Towards that end, the last fifteen years have witnessed the publication of several algorithms for an IR based approach to solving this problem. An examination of these prior contributions reveals that (1) They mostly used datasets of relatively small sizes for the evaluation of the proposed algorithms; and (2) The datasets used consisted mostly of Java-based projects.

To elaborate on the dataset sizes used in the prior studies, at the low end, the researchers have reported results using just a few hundred bug reports, and, at the high end, the reported results were based on just a few thousand bug reports. The studies presented in [16], [43], and [35] are the only ones that include more than a few thousand queries to evaluate the performance of their algorithms.

Regarding the above-mentioned studies that are based on large datasets, Ye et al. [43] evaluated their bug localization algorithm on around 20,000 bug reports drawn from six Java projects. The study presented in [35] was performed on 8000 bug reports belonging to three Java and C/C++ based projects. The most recent large-scale comparative study carried out by Lee et al. [16] used around 9000 bug reports, all belonging to Java-based projects. These three studies, however, evaluate bug localization methods belonging only to the first and the second generations of tools, and are mostly focused toward Java based projects. Therefore, a large-scale bug localization study that involves code libraries in multiple languages and that includes all three generation of tools has yet to be carried out. The goal of our paper is to remedy this shortcoming.

In this paper we present a comprehensive large-scale evaluation of a representative set of IR-based bug localization tools with the set spanning all three generations. The evaluation dataset we use, named Bugzbook, consists of over 20,000 bug reports drawn from a diverse collection of Java, C/C++, and Python software projects at GitHub. A large-scale evaluation such as the one we report here is important because it is not uncommon for the performance numbers produced by testing with a large dataset to be different from those obtained with smaller datasets.

An important issue related to any large-scale evaluation is the quality of the evaluation dataset — in our case, that would be the quality of the bug reports — to make sure that the dataset does *not* include duplicate bug reports and other textual artifacts that are not legitimate bug reports. Our Section 4.2 describes how the raw data was filtered in order to retain only the legitimate and non-duplicate bug reports.

For the large-scale evaluation reported here, we chose eight search tools, one from each generation of the now 15-year history of the development of such tools. As mentioned previously, the earliest of the tools — the first-generation tools — are based solely on BoW modelling in which the relevance of a file to a bug report is evaluated by comparing the frequencies of the terms appearing in the file with the frequencies of the terms appearing in the bug report. In general, a BoW approach may either be deterministic or probabilistic. For the deterministic versions of such tools, we chose the TFIDF (Term Frequency Inverse Document Frequency) approach presented in [28]. And, for probabilistic BoW, we chose what is known as the FI (Full Independence) version of the framework based on Markov Random Fields (MRF) [19, 31]. The probabilistic version is also referred to as the Dirichlet Language Model (DLM) [46].

For representing the second generation tools, we chose BugLocator [47] and BLUiR (Bug Localization Using information Retrieval) [30]. In addition to the term frequencies, these tools also exploit the structural information (when available) and information related to the revision history of a software library.

That brings us to the third generation tools that, in addition to the usual term frequencies, also take advantage of term-term order and contextual semantics in the source-code files, on the one hand, and in the bug reports, on the other. We have used the algorithms described in [31] and [1] to represent this generation of tools.

In addition to generating the usual performance numbers for the algorithms chosen, our large-scale evaluation also provides answers to the six research questions that are listed in Section 5.3 of this paper. Most of these questions deal with the relative importance of the different components of the algorithms that belong to the second and the third generation of the tools.

At this point, the reader may ask: What was learned from our large-scale multi-generational evaluation that was not known before? To respond, here is a list of the new insights we have gained through our study:

(1) Contrary to what was reported earlier, the retrieval effectiveness of two different ways of capturing the term-term dependencies [31] in software libraries — these are referred to as MRF-SD and MRF-FD — is the same.
(2) The performance of second generation tools BugLocator and BLUiR are not equivalent in terms of retrieval precision, contradicting the finding presented in [16].
(3) Including software libraries in different languages (Java, C++, and Python) in our study has led to a very important new insight: for the contextual semantics needed for the third-generation tools, it is possible to use the word embeddings generated for one language for developing a bug localization tool in another language. We refer to this as the "cross-utilization of word embeddings."

Note that these are just the high-level conclusions that can be made from the answers to the six questions listed in Section 5.3.

2 A TIMELINE OF PAST STUDIES IN IR-BASED BUG LOCALIZATION

A timeline of important publications on the subject of automatic bug localization is presented in Figure 1. The figure shows around 30 papers published between the years 2004 and 2019. These publications that appeared in roughly 15 highly-respected venues — conferences and journals — belong to the three generations of software bug localization. The names of these conferences and journals are also shown in the figure.

From 2004 to 2011 — that's when the first-generation tools came into existence — one could say that research in automatic bug localization was in its infancy. The algorithms presented in [13, 17, 18, 27] laid the foundations for such tools and these were based purely on the Bag-of-Words (BoW) based assumption. Marcus et al. [18] led the way through their demonstration that Latent Semantic Indexing (LSI) could be used for concept location. Kuhn et al. [13] extended the work of Marcus et al. and presented results in software comprehension. Next came the Latent Dirichlet Allocation (LDA) based bug localization algorithm proposed by Lukins et al. [17]. To round off this series of algorithms, Rao and Kak [27] compared several early BoW based IR techniques for bug localization, and showed that simpler BoW based approaches, such as Vector Space Model (VSM) and Unigram Model (UM) outperformed the more sophisticated ones, such as those using LDA.

The second-generation bug localization tools, developed between the years 2010 and 2016 [8, 9, 21, 23, 30, 33, 38–40, 42, 45, 47], exploit structural information embedded in the source code files and in the bug reports as well as the software-evolution related information derived from bug and version histories to enhance the performance of BoW based systems. These studies suggest that the information derived from the evolution of a software project such as historical bug reports [9, 23, 42, 47] and code change [33, 38, 39, 45] history plays an important role in localizing buggy files given a bug report. These studies also suggest that exploiting structural information embedded in the source code files [30, 38, 40, 45], such as method names and class names, and in the bug reports [8, 21, 40, 45], such as execution stack traces and source code patches, enhances the performance of a bug localization system. BugLocator [47], DHbPd (Defect History based Prior with decay) [33], BLUiR (Bug Localization Using information Retrieval) [30], BRTracer (Bug Report Tracer) [40], LOBSTER (Locating Bugs using Stack Traces and text Retrieval) [21], Amalgam (Automated Localization of Bug using Various Information), [38], BLIA (Bug Localization using Integrated Analysis) [45], and LOCUS (LOcating bugs from software Change hUnkS) [39] are some of the prominent bug localization tools developed during the second-generation.

The third and the most recent generation of bug localization tools date back to roughly 2016 when term-term order and semantics began to be considered for improving the retrieval performance of such tools [1, 22, 31, 36, 44]. For exploiting the term-term order, as for example reported in [31], these tools utilized the Markov modeling ideas first advanced in the text retrieval community [19]. And for incorporating contextual semantics, as in [1, 22, 36, 44], the tools used word embeddings based on the *word2vec* modelling [20] of textual data.

For the sake of completeness, it is important to point out that the organization of our evaluation study resulted in our having to leave out the contributions in two additional and relevant threads of research: (1) the query reformulation based methods for bug localization, such as those reported in [26, 34] and (2) the machine-learning and deep-learning based methods [10, 12, 14, 35, 41, 42, 44] in which a ranking model is trained to produce relevance scores for source code files vis-a-vis historical bug reports, and afterwards, the

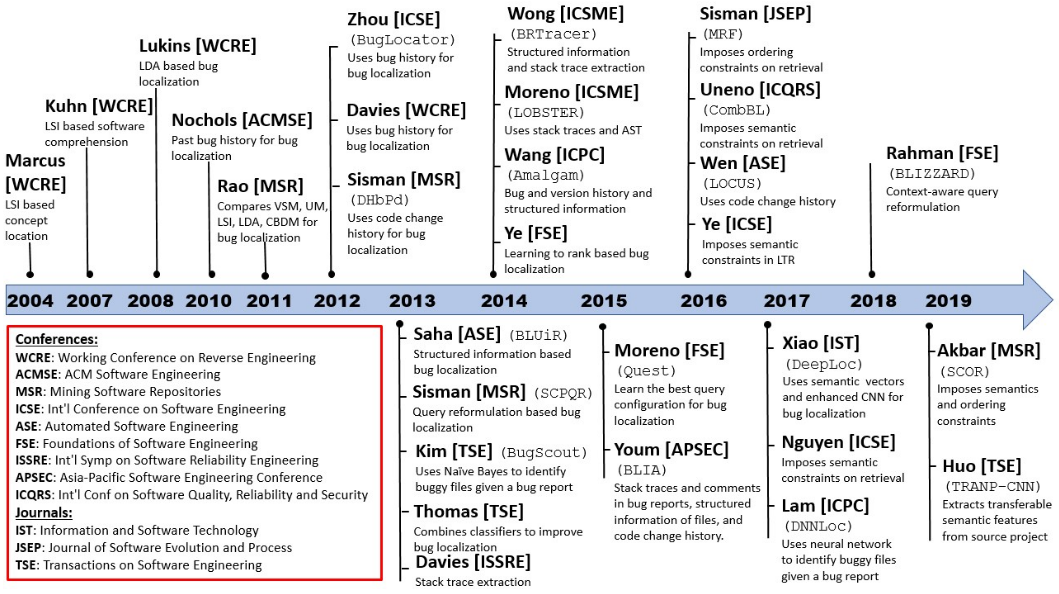

Marcus [WCRE]
LSI based concept location
Kuhn [WCRE]
LSI based software comprehension
Lukins [WCRE]
LDA based bug localization
Nochols [ACMSE]
Past bug history for bug localization
Rao [MSR]
Compares VSM, UM, LSI, LDA, CBDM for bug localization
Zhou [ICSE]
(BugLocator)
Uses bug history for bug localization
Davies [WCRE]
Uses bug history for bug localization
Sisman [MSR]
(DHbPd)
Uses code change history for bug localization
Wong [ICSME]
(BRTracer)
Structured information and stack trace extraction
Moreno [ICSME]
(LOBSTER)
Uses stack traces and AST
Wang [ICPC]
(Amalgam)
Bug and version history and structured information
Ye [FSE]
Learning to rank based bug localization
Sisman [JSEP]
(MRF)
Imposes ordering constraints on retrieval
Uneno [ICQRS]
(CombBL)
Imposes semantic constraints on retrieval
Wen [ASE]
(LOCUS)
Uses code change history
Ye [ICSE]
Imposes semantic constraints in LTR
Rahman [FSE]
(BLIZZARD)
Context-aware query reformulation
2004
2007
2008
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
Conferences:
WCRE: Working Conference on Reverse Engineering
ACMSE: ACM Software Engineering
MSR: Mining Software Repositories
ICSE: Int'l Conference on Software Engineering
ASE: Automated Software Engineering
FSE: Foundations of Software Engineering
ISSRE: Int'l Symp on Software Reliability Engineering
APSEC: Asia-Pacific Software Engineering Conference
ICQRS: Int'l Conf on Software Quality, Reliability and Security
Journals:
IST: Information and Software Technology
JSEP: Journal of Software Evolution and Process
TSE: Transactions on Software Engineering
Saha [ASE] (BLUiR)
Structured information based bug localization
Sisman [MSR] (SCPQR)
Query reformulation based bug localization
Kim [TSE] (BugScout)
Uses Naïve Bayes to identify buggy files given a bug report
Thomas [TSE]
Combines classifiers to improve bug localization
Davies [ISSRE]
Stack trace extraction
Moreno [FSE]
(Quest)
Learn the best query configuration for bug localization
Youm [APSEC]
(BLIA)
Stack traces and comments in bug reports, structured information of files, and code change history.
Xiao [IST]
(DeepLoc)
Uses semantic vectors and enhanced CNN for bug localization
Nguyen [ICSE]
Imposes semantic constraints on retrieval
Lam [ICPC]
(DNNLoc)
Uses neural network to identify buggy files given a bug report
Akbar [MSR]
(SCOR)
Imposes semantics and ordering constraints
Huo [TSE]
(TRANP-CNN)
Extracts transferable semantic features from source project

Figure 1: A 15-year timeline of the cited publications in the field of IR-based automatic bug localization. We represent a publication by the last name of the first author of the publication along with the venue in which the publication appeared. The abbreviation inside square brackets, for example "WCRE" in "[WCRE]", refers to the conference or the journal in which the publication appeared, while the abbreviation inside round brackets, for example "BLUiR" in "(BLUiR)" indicates the name of the tool presented in the publication. The list of publications mentioned in the timeline is, obviously, not complete and is only a representative subset of the hundreds of publications on IR-based bug localization. Notice that we have only included those publications in this timeline in which bug localization experiments were performed (with the exception of a few very early publications — that appeared before the year 2010, such as [18] — which performed experiments for concept location).

learned model is used to test the relevance of a new bug report to a source code file. We leave the evaluation of such bug localization methods for a future study.

With regards to a large-scale comparative evaluation, we are aware of only one other recent study [16] that evaluates six different IR based bug localization tools on a dataset that involves 46 different Java projects that come with 61,431 files and 9,459 bug reports. As the authors say, their work was motivated by the current "lack of *comprehensive evaluations* for state-of-the-art approaches which offer insights into the actual performance of the techniques." However, this study only covers bug localization methods from the second-generation of the tools, and therefore, does not include the important developments in bug localization made possible by the third-generation tools. That is, this study has left out the tools that incorporate term-term order and contextual semantics to enhance bug localization performance as in [1, 22, 31, 36, 44].

Additionally, note that the study carried out by Lee et al. [16] considers only Java-based software projects. On the other hand, our Bugzbook dataset based large-scale evaluation involves nine different IR tools from all the three generations of software bug localization systems, and is based on a diverse collection of Java, C/C++, and Python based software projects that come with 4.5 million files and over 20,000 bug reports.

For yet another reason as to why we did not use the Bench4BL toolchain, that toolchain was designed to work with the Jira issue tracking platform. Because of our interest in cross-language effects on retrieval platforms, we also wanted to download and process the bug reports from GitHub.

We should also mention the past studies by Ye et al. [43] and Thomas et al. [35] in which number of queries analysed are around 20,000 and 8,000, respectively. However, these studies are also focused mainly toward Java-based projects, and also do not consider the tools from the most recent generation of tools that include term-term order and semantics.

3 CATALOG OF THE BUG LOCALIZATION TOOLS IN OUR EVALUATION

The comparative evaluation we report in this paper involves the following bug localization tools:

1. TFIDF: TFIDF (Term Frequency Inverse Document Frequency) [28] works by combining the frequencies of query terms in the file (TF) and the inverse document frequencies of the query terms in the corpus (IDF) to determine the relevance of a file to a query.

2. DLM: DLM (Dirichlet Language Model) [31, 46] or FI (Full Independence) BoW is a probabilistic model that estimates a smoothed first order probability distribution of the query terms in the file to produce the relevance score for the file given the query.

3. BugLocator: BugLocator [47] takes into account the history of the past bug reports and leverages similar bug reports

Table 1: Comparison of the different bug localization tools based on the logic components used for ranking files.

	TI	DLM	BL	BR	MRF		SCOR	
					SD	FD	PWSM	SCOR
BoW	✓	✓	✓	✓	✓	✓	✓	✓
Order					✓	✓		✓
Semantic							✓	✓
Trace					✓	✓	✓	✓
Structure				✓				
Past Bugs			✓					

TI - TFIDF, DLM - Dirichlet LM, BL - BugLocator, BR - BLUiR

that have been previously fixed to improve bug localization performance.

4. BLUiR: BLUiR (Bug Localization Using information Retrieval) [30] extracts code entities such as classes, methods, and variable names from source code files to help in localizing a buggy file.

5. MRF SD: MRF (Markov Random Field) based SD (Sequential Dependence) model [31] measures the probability distribution of the frequencies of the pairs of *consecutively* occuring query terms appearing in the file to compute the relevance score for the file given a query.

6. MRF FD: MRF based FD (Full Dependence) [31] is a term-term dependency model that considers frequencies of *all* pairs of query terms appearing in the file to determine the relevance of the file to the query.

7. PWSM: PWSM (Per-Word Semantic Model) [1] uses word embeddings derived from the word2vec algorithm to model term-term contextual semantic relationships in retrieval algorithm.

8. SCOR: SCOR (Source code retrieval with semantics and order) [1] combines the MRF based term-term dependency modeling, as described in [31], with semantic word embeddings as made possible by word2vec [20] to improve bug localization performance.

In Table 1, we compare the bug localization tools in our evaluation based on the logic components they use to produce the relevance score for a file vis-a-vis a given bug report.

4 BUGZBOOK: A LARGE DATASET FOR RESEARCH IN SOFTWARE SEARCH

The Bugzbook dataset we have used for the comparative evaluation reported in this paper consists of over 20,000 bug reports. That makes it one of the largest datasets for research in software search, in general, and automatic bug localization in particular. Table 2 compares Bugzbook with several other bug localization datasets.

The over 20,000 bug reports in Bugzbook were drawn from 29 projects. For each project, we kept track of the associations between the bug reports and the version number of the project for which the bug was reported. We believe Bugzbook will encourage researchers to carry out large-scale evaluations of their software retrieval algorithms. Given the size of the dataset, it may also encourage further research in deep-learning based approaches for software search.

Table 2: Comparing Bugzbook with other datasets

Dataset	#projects	# bugs reports
moreBugs	2	~400
BUGLinks	2	~4000
iBUGS	3	~400
Bench4BL	46	~10000
Bugzbook	29	~20000

In the subsection that follows, we highlight some unique features of Bugzbook. In a later subsection, we then explain the process that was used to construct this dataset.

4.1 Features of Bugzbook

As shown in Table 3, the Bugzbook dataset includes a large collection of Java, C/C++, and Python projects, 29 to be exact. The reader should note that two of the Java based projects listed in the table, AspectJ and Eclipse, were used previously in two datasets, iBugs [37] and BUGLinks [32], that have frequently been used for testing new algorithms for automatic bug localization.

Bugzbook includes several Apache projects. The reason for selecting projects from Apache is because its software developer community is believed to be the largest in the open-source software world with regards to Java programming language. From Apache we only selected those projects for which we could find the bug reports online in the well managed Jira [5] issue tracking platform.

In addition to the Apache projects, Bugzbook also contains bug reports from other large-scale open-source Projects, such as Tensorflow, OpenCV, Chrome, and Pandas. The bug reports for these projects are maintained on the GitHub platform[1].

As shown in Table 3, the total number of bug reports in Bugzbook is 21,253. The total number of source-code files in all of the projects together adds up to 4,253,610. Note that the last column of the table shows the number of versions for each project. As mentioned earlier, we maintain the association between the bug reports and the project versions they belong to. Finally, the data format used in Bugzbook for storing the bug reports is the same XML schema as used previously for BugLinks.

4.2 How the Bugzbook Dataset was Constructed and, More Importantly, Sanitized

The Bugzbook dataset was constructed from open-source software repository archives and their associated issue tracking platforms. The Apache project archive and the associated Jira issue tracking platform would be prime examples of that.

In the material that follows in this subsection, we will address the following steps used to create Bugzbook: (1) Gathering the raw data for bug reports and source code files; (2) Filtering the raw bug reports to eliminate any duplicates and other textual artifacts; (3) Linking the bug reports with their respective source code files after the files were fixed; (4) Matching each bug report with the respective project version; and, finally, (5) carrying out a manual

[1]Chrome bug reports are obtained from BUGLinks website.

Table 3: Stats related to Bugzbook dataset.

Project	Description	# files	# bugs	# vers
Java projects				
Ambari	Hadoop cluster mgr	85113	2253	29
Aspectj	Java extension	6636	291	1
Bigtop	Big data manager	1291	5	5
Camel	Integration library	1229503	2308	101
Cassandra	Database mgmt tool	187150	514	133
Cxf	Services framework	768444	1795	138
Drill	Hadoop query	42360	800	17
Eclipse	IDE	12825	4035	1
HBase	Database mgmt tool	265491	2476	95
Hive	Data warehouse	114993	2221	32
JCR	Content Repository	472680	457	104
Karaf	Server-side app	63420	390	34
Mahout	Machine learning	27263	162	10
Math	Mathematics tool	16735	17	3
OpenNLP	NLP library	10250	84	11
PDFBox	PDF processor	38943	1163	35
Pig	Database manager	25462	47	11
Solr	Search server	404944	471	54
Spark	Database manager	18737	185	29
Sqoop	Database manager	7415	201	7
Tez	Graph processor	14795	177	14
Tika	Docs processor	16983	183	16
Wicket	Web app	317975	567	63
WW	Web app	72838	87	23
Zookeeper	Distr comp tool	9911	20	9
C/C++ and Python projects				
Chrome	Browser	7232	147	1
OpenCV	Computer vision tool	2865	8	1
Pandas	Data analysis tool	523	179	1
Tensorflow	Deep learning tool	10833	10	1
	Total	**4253610**	**21253**	**976**

verification of the dataset on randomly chosen bug reports and the corresponding source code files.

4.2.1 Gathering Raw Bug Reports and Source Code Files. Jira, the issue tracking platform for Apache, provides bug reports in XML format with multiple fields. We wrote a script that automatically downloaded all the bug reports that were marked as "FIXED" by the issue tracker and stored them in a disk file. The reason we downloaded only the fixed bug reports is because we could obtain the relevant source code files that were fixed in response to those bugs. With regard to downloading bug reports from GitHub, we used a publicly available Python script [7]. We modified the script so that it downloaded only those reports from GitHub that were explicitly marked as "closed bugs" by the report filer. This overall approach to the creation of an evaluation dataset has also been used in the past for creating some well-known other datasets [16, 31, 47].

That brings us to the downloading of the source-code file. For downloading these files for the Apache projects, we wrote another script that automatically downloaded all the versions of the software projects we use in this study from the Apache archives website [3]. These software repositories were downloaded in the form of compressed ZIP or TGZ archives. The compressed files belonging to the different versions of the projects were then extracted from the archives and stored in the disk.

In addition to downloading the archives for the software projects, we also cloned the most recent snapshot of the projects from the relevant version control platforms (GitHub, GitBox, etc.) in order to obtain the most recent commit logs for the software repositories. As explained later in this section, the commit logs are used to establish associations between the bug reports and the files.

As for the Eclipse, Chrome, and AspectJ projects, we downloaded their bug reports from the BUGLinks and the iBUGS datasets that are available on the internet. Since these bug report relate to a single version of the project, we downloaded just those versions from the Eclipse, Chrome, and AspectJ archived project repositories.

4.2.2 Filtering the Raw Bug Reports. On the Jira online platform [4], the individual filing a report has the option to label it as belonging to one of the following categories: "task", "subtask", "story", "epic", or "bug". We made sure that we downloaded only those reports that were labeled "bug" for the Bugzbook dataset.

On GitHub as well, the individual filing a report has the option to assign labels to the report based on pre-defined categories[2]. We select only those reports for Bugzbook that had been marked explicitly as "bug" or "Bug" by whomsoever filed the reports.

Finally, in order to avoid including duplicate bug reports in the Bugzbook dataset, we only selected those bug reports that were *not* marked as a "duplicate" of another bug report by the report filer.

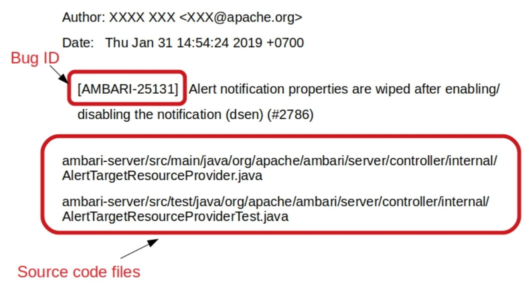
4.2.3 Linking Bug Reports with Source Code Files. The most difficult part of what it takes to create a dataset like Bugzbook is the linking of the bug reports with the source code files which were fixed in response to the bug reports. This step is critical because it provides the ground truth data with which a bug localization technique can be evaluated.

The commit messages that are filed when the developers modify or fix the files play an important role in linking the bug reports with the relevant files. If a commit is about a bug having been resolved, the developer who fixed the bug includes in the commit message the ID of the bug that was fixed as a specially formatted string. For most of the projects we examined, this string is in the following format: "PROJECT-###", where "PROJECT" is the name of the software project, such as "AMBARI", and "###" is the ID of the bug report that was resolved. An example of a commit message with the bug ID and the names of the source code files fixed is shown in the Figure 2.

A GIT based version control system that manages a software project also attaches the names of the files that were modified in response to a bug report with the commit messages. The associations thus created between the file names and the bug reports can be used directly to link the bug reports with the relevant source code files.

Although there are advanced techniques available in the software engineering literature [15] that automatically link bug reports with source code files on the basis of textual information contained in the bug reports and the commit messages, we use the explicit method

[2]These categories are defined by the project administrators


Author: XXXX XXX <XXX@apache.org>

Date: Thu Jan 31 14:54:24 2019 +0700

[AMBARI-25131] Alert notification properties are wiped after enabling/ disabling the notification (dsen) (#2786)

ambari-server/src/main/java/org/apache/ambari/server/controller/internal/AlertTargetResourceProvider.java

ambari-server/src/test/java/org/apache/ambari/server/controller/internal/AlertTargetResourceProviderTest.java

Figure 2: A commit message with bug ID and source code files highlighted in the text.

described above to establish the links between the bug reports and the files. To elaborate further, by explicit we mean that if a commit message mentions a file name along with the bug ID, then we can match up the two and form a link. Otherwise, we discard the commit message. The reason to use this explicit method for linking the bug reports with the source code files is because we want to avoid false positives in the linking process at the possible cost of incurring false negatives.

4.2.4 Versioned Associations between the Bug Reports and the Files. In much research in the past on automatic bug localization, the practice was to use only the latest version of the software library for the source code and for file identification. Bugzbook, on the other hand, maintains all of the different versions of a software project and the files relevant to a bug belong to a specific version of the project.

The bug reports often come with either the version number of the software that is presumably the source of the bug, or the version number in which the bug is fixed. If the affected version of the project that resulted in a bug is present in the bug report description, we link the bug report with the version mentioned in the report. On the other hand, if the bug report mentions the fixed version of the software, we use the version that was released prior to the fixed version as the linked version for the bug report. This obviously is based on the assumption that the version that was released prior to the fixed version contained the bug mentioned in the bug report.

4.2.5 Manual Verification of the Bugzbook Dataset. For verification of the steps described previously in this section to control the quality of the dataset, we manually check a randomly chosen small portion of the dataset by comparing the bug report entry in the Bugzbook dataset with the bug report entry in the online bug tracking platform like Jira and GitHub. In particular, we randomly selected two bug reports from each software project present in Bugzbook and manually verified its entry in the online platform. We check if the bug ID associated with a bug report in Bugzbook indeed belongs to the correct bug report in the online tracking system. We also verify all the attributes, such as title and description entries, of the bug reports. In addition to verifying the bug report entry in the online tracking system, we also verify if the bug ID associated with the bug report has a commit message associated with it in the GIT commit log, and that the fixed files mentioned in the commit log match the repaired files stored in the Bugzbook entry of the bug report.

5 EXPERIMENTAL RESULTS

This section presents the experimental results of our large-scale study on the effectiveness of modeling source code repositories using first, second, and third generations of bug localization tools. Also presented in this section is the evaluation metric used.

5.1 Implementation Details

For the first generation tools — DLM (FI BoW) and TFIDF — we used the implementations provided by the popular open-source search engine tool Terrier [24].

For the second generation tools we used the implementations of BugLocator [11], and BLUiR [29] that have been made available online by the authors of the tools. For these tools, we used the parameter settings as suggested by the same authors.

For the third generation tools, for MRF-SD and MRF-FD we used the implementations that are built into Terrier engine. And for PWSM and SCOR, we used the implementation provided by the authors of [1]. This implementation uses the DLM-FI BoW model as the baseline model upon which enhancements are made to introduce the semantic and ordering relationships between the terms. For the word embeddings needed by PWSM and SCOR, these were downloaded from the website where the authors of SCOR [1] have posted the embeddings for half a million software terms [2].

We use the parameters recommended by the authors of the respective tools to evaluate their performance on bug localization dataset.

5.2 Evaluation Metrics

We use the Mean Average Precision (MAP) values to evaluate the performance of retrieval algorithms. This metric is the mean of the Average Precisions (AP) calculated for each of the bug report queries. The MAP values are subject to statistical significance testing using the Student's Paired t-Test. Significance testing tells us whether the measured difference in the results obtained with two different retrieval models is statistically significant. Student's t-Test has been used in previous studies [31, 47] to establish the performance gain of one algorithm over another.

5.3 Retrieval Experiments

We provide bug localization results for comparing the following eight retrieval algorithms: (1) TFIDF, (2), FI BoW, (3) BugLocator, (4) BLUiR, (5) MRF SD, (6) MRF FD, (7) PWSM, and (8) SCOR. Through our retrieval experiments we attempt to answer the following 6 important research questions:

RQ1: In terms of retrieval precision, how do the first, second, and third generation tools compare against each other?

RQ2: Does the performance of the retrieval algorithms depend on the programming language used in the software?

RQ3: Are the word embeddings provided by SCOR really generic?

RQ4: How to best create a composite retrieval performance metric for large-scale evaluations?

RQ5: Does changing the semantic word embeddings affect the performance of the semantics-based retrieval algorithms?

Table 4: MAP values for the retrieval algorithms evaluated on Bugzbook dataset. Notice that the last two rows contain the MAP values for the eight retrieval algorithms averaged across all projects, and the Mutual Information weighted (MI-wtd) MAP values for the same, respectively. Also notice that the second column contains the MI values for each project. We also show in the second to the last row and, in the 8th row from the bottom for just the Java-based projects, the results of significance testing. The superscript denotes the significant difference when considering p-value less than 0.05, while the subscript denotes significance difference when considering p-value less than 0.01. For example, in the second to the last row of MRF SD column, we have 0.370^{tdlfp}_{tdlp} which specifies that MRF SD is significantly better than TFIDF, DLM, Buglocator, MRF FD, and PWSM when considering p-value less than 0.05, while it is significantly better than only TFIDF, DLM, BugLocator, and PWSM when considering p-value less than 0.01.

Project	MI	TFIDF	DLM	BugLocator	BLUiR	MRF SD	MRF FD	PWSM	SCOR
Ambari	1.98	0.268	0.227	0.257	0.242	0.268	0.278	0.253	0.295
Aspectj	1.07	0.211	0.216	0.220	0.250	0.226	0.230	0.233	0.250
Bigtop	0.47	0.456	0.079	0.080	0.567	0.304	0.300	0.110	0.560
Camel	1.90	0.390	0.369	0.345	0.345	0.405	0.382	0.395	0.407
Cassandra	1.69	0.364	0.394	0.361	0.394	0.367	0.310	0.356	0.411
Cxf	1.46	0.332	0.287	0.319	0.303	0.348	0.342	0.329	0.363
Drill	1.58	0.196	0.210	0.170	0.169	0.218	0.189	0.223	0.240
Eclipse	1.30	0.284	0.248	0.310	0.320	0.303	0.305	0.271	0.320
HBase	1.74	0.387	0.362	0.370	0.333	0.429	0.424	0.408	0.453
Hive	1.77	0.332	0.278	0.219	0.224	0.335	0.346	0.269	0.345
JCR	1.88	0.432	0.396	0.417	0.394	0.453	0.454	0.437	0.450
Karaf	1.80	0.372	0.386	0.348	0.382	0.374	0.332	0.399	0.427
Mahout	1.27	0.320	0.295	0.481	0.367	0.315	0.267	0.320	0.338
Math	0.81	0.482	0.495	0.601	0.557	0.454	0.481	0.458	0.512
Opennlp	1.06	0.435	0.456	0.500	0.261	0.433	0.347	0.437	0.498
PDFBox	1.57	0.351	0.319	0.430	0.370	0.368	0.357	0.358	0.380
PIG	0.85	0.285	0.228	0.360	0.315	0.295	0.312	0.311	0.335
SOLR	1.20	0.343	0.305	0.323	0.331	0.371	0.370	0.344	0.394
Spark	1.77	0.339	0.369	0.398	0.348	0.377	0.332	0.362	0.418
Sqoop	1.40	0.358	0.385	0.379	0.406	0.367	0.307	0.322	0.417
Tez	1.48	0.373	0.373	0.376	0.277	0.424	0.428	0.401	0.431
Tika	1.27	0.341	0.270	0.375	0.411	0.290	0.316	0.333	0.326
Wicket	1.99	0.439	0.420	0.489	0.411	0.450	0.389	0.399	0.440
WW	1.34	0.397	0.354	0.288	0.226	0.414	0.379	0.376	0.430
Zookeeper	0.99	0.468	0.494	0.502	0.456	0.565	0.527	0.532	0.529
Average MAP (Java)		$\mathbf{0.358}^{dr}_{dr}$	**0.329**	$\mathbf{0.357}^{drf}_{dr}$	$\mathbf{0.346}^{d}$	$\mathbf{0.366}^{tdlrfp}_{tdlrp}$	$\mathbf{0.348}^{dr}_{dr}$	$\mathbf{0.345}^{d}_{d}$	$\mathbf{0.399}^{tdlrsfp}_{tdlrsfp}$
Chrome	0.58	0.113	0.118	0.039	-	0.119	0.101	0.122	0.137
OpenCV	0.16	0.481	0.802	0.195	-	0.845	0.680	0.818	0.819
Pandas	0.64	0.266	0.265	0.266	-	0.375	0.405	0.388	0.435
Tensorflow	0.23	0.208	0.166	0.111	-	0.246	0.163	0.189	0.182
Average MAP (C/C++/Python)		**0.267**	**0.338**	**0.153**	-	**0.396**	**0.339**	**0.379**	**0.393**
Average MAP (Overall)		$\mathbf{0.346}^{d}_{d}$	**0.330**	**0.328**	-	$\mathbf{0.370}^{tdlfp}_{tdlp}$	$\mathbf{0.347}^{d}_{d}$	$\mathbf{0.350}^{df}_{df}$	$\mathbf{0.398}^{tdlsfp}_{tdlsfp}$
MI-wtd MAP		**0.447**	**0.424**	**0.447**	-	**0.467**	**0.442**	**0.446**	**0.500**

t: > TFIDF d: > DLM (FI) l: > BugLocator r: > BLUiR s: > MRF SD f: > MRF FD p: > PWSM

RQ6: Does replacing DLM with TFIDF in MRF based frameworks enhance the performance of bug localization systems?

The questions RQ1 and RQ2 are important because they represent the primary motivation for our research. As for RQ3, RQ5, and RQ6, they are included because of the current focus of research in software mining and text retrieval, which is exploiting semantics and term-term ordering for retrieval. Finally, RQ4 reflects moving from small-scale evaluations to large-scale evaluations.

The MAP performance numbers for the eight retrieval algorithms evaluated on 29 Java, C/C++, and Python projects present in Bugzbook are shown in Table 4. In the discussion that follows, we use this table to answer the six important research questions posed above.

Regarding the empty entries in the last six rows of the BLUiR column in Table 4, since this tool was designed specifically for Java source code, we do not report on its performance on non-Java projects (these being Chrome, OpenCV, Pandas, and Tensorflow). BLUiR uses a Java-specific parser to extract the method, the class, and the identifier names, and the comment blocks from Java source code files. Therefore, in all our comparison involving BLUiR, we include only the Java based projects in Bugzbook.

RQ1: In terms of retrieval precision, how do the first, second, and third generation tools compare against each other?

TFIDF and DLM are the two first generation tools whose average MAP values across all software projects in Bugzbook are 0.346 and 0.330, respectively, as shown in the table. Our results show that TFIDF outperforms DLM (or FI BoW) model by around 5%. The performance difference between TFIDF and FI is significant even when considering p-value less than 0.01. This implies that when considering pure-BoW based tools one should choose TFIDF over FI (DLM) model.

The second generation tools BugLocator and BLUiR that incorporate software-evolution history and structural information have average MAP values across all Java projects of 0.357 and 0.346, respectively. The MAP value for BugLocator on all the projects present in Bugzbook is 0.328. Both these bug localization tools perform significantly better than the FI BoW (DLM) model when only Java projects are considered in evaluation and when p-value is 0.05. However, when p-value is 0.01, only BugLocator outperforms DLM. The performance numbers for BugLocator and DLM when all the projects in Bugzbook (including Java, C/C++, and Python projects) are considered are comparable.

We note that the simple TFIDF BoW model significantly outperforms BLUiR by 4% when examined through our large-scale bug localization study of Java projects. In a project-by-project comparison, BLUiR outperforms TFIDF in just 12 out of the 25 Java-based projects in Table 4 . Amongst these, the comparative results for AspectJ and Eclipse are along the same lines as those reported previously in the original BLUiR paper. However, with regard to the projects on which BLUiR was not evaluated previously, its performance on several Apache based projects is worse than that of TFIDF.

On the other hand, the performance numbers for TFIDF and BugLocator are comparable. The performance of BugLocator is significantly better than that of BLUiR for the Java only projects. This contradicts the finding presented in [16] and [30].

The third generation order-only MRF SD and MRF FD models with average MAP values across all projects of 0.370 and 0.347, significantly outperform the first generation tool DLM by 12% and 5%, respectively. This confirms the finding in [31]. However, when compared with TFIDF, while MRF SD significantly outperforms TFIDF, the performance of MRF FD is similar to that of TFIDF.

We observe that the two order-only MRF SD and MRF FD retrieval models perform equivalently when evaluated using statistical t-testing and considering p-value less than 0.01. This contradicts the finding in [31] which shows that the performance of MRF SD and MRF FD are similar in terms retrieval accuracy.

We notice that MRF SD outperforms MRF FD on 19 out of 29 projects. The projects on which MRF FD outperforms MRF SD are Ambari, AspectJ, Eclipse, Hive, JCR, Math, Pig, Tez, Tika, and Pandas. Most of these projects have large number of bug reports and contribute in total around 10000 — that is roughly around 50% — of bug reports to the Bugzbook dataset. Since both MRF SD and MRF FD outperform each other on roughly equal number of bug reports, this is a possible reason for their statistically equivalent performance.

We compare the performance of second generation tools, BugLocator and BLUiR, with the pure-ordering based third generation tools, MRF SD and MRF FD, and observe that both MRF SD and MRF FD outperform both BugLocator and BLUiR.

The performance of MRF SD is significantly better than that of both BugLocator and BLUiR on Java based projects. MRF SD also significantly outperforms BugLocator by 13% on all the projects in Bugzbook. The performance of MRF FD is better than that of BLUiR. The performance of BugLocator is better than that of MRF FD on Java projects with a p-value of 0.05. However, the performance numbers for the two are comparable when p-value of 0.01 is considered. Their performance is also comparable when all projects in Bugzbook are considered. This result contradicts the results reported in [31].

When considering semantics-only based retrieval with the PWSM model we observe a mean MAP value of 0.350 across all the projects in the Bugzbook dataset. We notice that whereas PWSM outperforms DLM significantly by 6%, it does not do so vis-a-vis TFIDF. The performance of PWSM is comparable to that of BLUiR when only the Java projects are considered. Additionally, PWSM does not significantly outperform BugLocator when all projects in Bugzbook are considered. The percentage difference between the all-projects performance numbers for PWSM and BugLocator is around 6%.

The performance of PWSM — which is a pure-semantics based third generation tool — is comparable to the performance of pure-ordering based MRF SD model. However, PWSM significantly outperforms MRF FD model. This comparison is not performed in [1]. The performance of SCOR — which combines MRF based term-term ordering dependencies with word2vec based semantic word embeddings, outperforms the first and second generation tools. We observe that the performance of SCOR is significantly better than the other seven retrieval algorithms when considering retrieval accuracies.

RQ2: Does the performance of the retrieval algorithms depend on the programming language used in the software?

In many past studies, only Java based software projects were used for evaluating the performance of bug localization tools. This question is important as it helps in determining the performance of these bug localization tools on non-Java projects. To answer this question we compare the performance of each retrieval algorithm on projects written in Java and other programming languages.

The average MAP values for all the eight retrieval algorithms on projects that only use Java programming language are shown in the 8th row from the bottom in Table 4. The average MAP values for all retrieval algorithms except BLUiR on C/C++ and Python based projects are shown in the 3rd row from the bottom in Table 4.

We notice that the performance of all retrieval algorithms except for TFIDF and BugLocator on Java-based libraries is similar to what we get on C/C++ and Python based projects. TFIDF and BugLocator perform significantly poorly on non-Java projects. We also observe that the semantics-based retrieval algorithms perform surprisingly very well on C/C++ and Python projects. What makes the last observation all the more surprising is that the word2vec algorithm was trained on only the Java based projects used by SCOR.

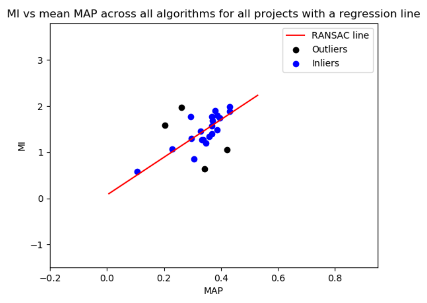


Figure 3: Scatter plot of average MAP vs MI values. Each data point in the plot is a tuple (MAP, MI) for a software project. The MAP value plotted for a software project is the mean of the 8 MAP values obtained while evaluating the 8 retrieval algorithms on a specific project. Also shown in the figure is the RANSAC fitted line along with inlier and outlier points. A low MI implies a difficult project, which in turn, implies a low mean MAP value for the retrieval algorithms.

With a minimum MAP value of 0.039 for BugLocator and a maximum MAP value of 0.137 for SCOR, Chrome is the project on which the performance of all the retrieval algorithms is the lowest. The top three algorithms on Chrome are SCOR, PWSM, and MRF SD with MAP values of only 0.137, 0.122, and 0.119, respectively.

The MAP values for all retrieval algorithms except for TFIDF and BugLocator on the 8 bug reports of the OpenCV project are very high. The three bug localization techniques that worked the best on the OpenCV project are MRF SD, SCOR, and PWSM with MAP values of 0.845, 0.819, and 0.818, respectively.

As for Pandas — a pure Python project — SCOR, MRF FD, and PWSM are the three algorithms that perform the best in terms of retrieval precision with MAP values of 0.435, 0.405, and 0.388.

The MAP values of the retrieval algorithms on Tensorflow are not very impressive. The lowest performing algorithm, BugLocator, achieved a MAP value of only 0.111, while the top performing algorithm, MRF SD, works with a MAP value of only 0.246. The top three algorithms for this project are MRF SD, TFIDF, and PWSM with MAP values of 0.246, 0.208, and 0.189.

RQ3: Are the word embeddings provided by SCOR really generic?
In the SCOR paper [1], we claimed that the SCOR word embeddings generated by the word2vec algorithm in that paper would be generic enough so that they could be used for carrying out semantic search in new libraries, that is, the libraries that were not used for generating the embeddings. However, in [1], this claim was supported with the results from just one library, AspectJ.

Our new results, as reported in this paper, provide further affirmation for that claim. The Java-based dataset that was used for training the word2vec algorithm in [1] did not include the following Apache projects in the Bugzbook dataset: Bigtop, OpenNLP, PDFBox, and Drill. The retrieval results for SCOR on these four projects as shown in Table 4 speak for themselves.

Further affirmation of our claim is provided by the C/C++ and Python based projects in Bugzbook. We observe that the performance of SCOR on roughly 150 Chrome bug reports and roughly 180 Pandas bug reports is the best among all the retrieval algorithms. Notice that Chrome is a pure C/C++ based project while Pandas is a pure Python based project. On the 8 OpenCV bug reports and the 10 Tensorflow bug reports, however, the performance of MRF SD is better than that of SCOR.

Therefore, in answer to this question, we can say that the word embeddings generated by the word2vec algorithm in SCOR are generic enough to be used for carrying out semantic search not only in Java based projects not seen in SCOR but also in C/C++ and Python projects.

RQ4: How to best create a composite retrieval performance metric for large-scale evaluations?
When a bug localization dataset involves multiple projects, it is unlikely that all the projects would present the same level of difficulty (LoD) to a retrieval engine. So, ideally, one should weight the performance numbers for the different projects with some measure of LoD for the individual projects. We have experimented with the information-theoretic idea of Mutual Information (MI) for the source-code library and the bug reports as a measure of retrieval LoD for the library. We characterize each project by two random variables X and Y, where X represents the vocabulary in the source code and Y represents the vocabulary in all the bug reports for that project. Now we can measure MI for any given project by $MI(X, Y) = H(X) + H(Y) - H(X, Y)$, where $H(X)$ and $H(Y)$ are the marginal entropies and $H(X, Y)$ the joint entropy. Note that $MI(X, Y)$ quantifies the amount of information that the two random variables X and Y share. So the higher the value of MI for a project, the more the bug reports can tell us about the project vocabulary and vice versa.

When we plot the MI value for each project against the mean of the MAP performance numbers obtained with the different retrieval algorithms for that project, we obtain the scatter plot shown in Figure 3. We also show in the figure a least-squares line fitted to the data points using the RANSAC (Random Sample and Consensus) algorithm along with the inlier and the outlier points. The slope of this line is 4.085 and the intercept 0.073.

The correlation that is present between MI and MAP implies that MI captures, albeit approximately, the level of retrieval difficulty for a given software library along with its bug reports. Note that at any given value of MI, we do not distinguish between the retrieval algorithms in terms of their performance values. Rather, we take a mean MAP value across all the algorithms to represent the performance of all the retrieval algorithms on the project that corresponds to MI value.

The second column of Table 4 shows the calculated MI values for the software projects in Bugzbook. Also, the last row of the table shows the MI-weighted MAP values averaged across all the projects for each retrieval algorithm. We observe that when the MI value for a project is high — as for example 1.90 for Camel and 1.99 for Wicket — the MAP values of the retrieval algorithms for that project are also high. The lowest MAP value observed for Camel is 0.345 and for Wicket is 0.389. On the other hand, with a low MI value of 1.57 for Drill project, the highest MAP value in that row is only 0.240 for SCOR algorithm.

RQ5: Does changing the semantic word embeddings affect the performance of the semantics-based retrieval algorithms?

Table 5: Shown are the MAP values obtained while changing the size of the semantic word vectors for the SCOR algorithm, evaluated on the Eclipse software project. Also shown are the MAP values obtained while replacing word2vec with other word embedding generators.

SCOR-V500	SCOR-V1000	SCOR-V1500
0.3191	0.3204	0.3193
SCOR$_{\text{skipgram}}$	**SCOR$_{\text{glove}}$**	**SCOR$_{\text{fasttext}}$**
0.3204	0.3192	0.3182

The word embeddings can be changed either by changing the sizes of the vector involved, or by using different embeddings altogether.

To address the question related to the sizes of the vectors, we varied the size of the word2vec representations and generated the retrieval results for the SCOR retrieval model. The results are presented in Table 5. We refer to the different versions of SCOR, with each version using vectors of a specific size, as SCOR-V500, SCOR-V1000, and SCOR-V1500. In this notation, SCOR-VN uses word embedding vectors of size N. The first row of Table 5 shows the retrieval results on the Eclipse dataset that contains 4000 bug reports with the different versions of SCOR. Based on these results, we conclude that the size used for the word embeddings has no significant impact on the retrieval performance. We chose Eclipse for this test as it has already been used in several previous studies related to bug localization.

We also compare the performance of SCOR when it is used with different types of word embeddings. In addition to word2vec, there are now two other well-known word embeddings: GloVe (Global Vector Representations) [25] and FastText [6]. When we replace the word2vec Skipgram model with GloVe and FastText in SCOR, the difference observed in terms of MAP performance on the 4000 bug reports of Eclipse dataset is negligible as shown in the second row of Table 5. In that table, SCOR$_{\text{skipgram}}$ refers to the original SCOR algorithm, and SCOR$_{\text{glove}}$ and SCOR$_{\text{fasttext}}$ refer to the versions of SCOR using GloVe and FastText word embeddings. For all the three word embedding algorithms we use the same input training dataset that is available at our SCOR website. For this study, we used embedding vectors of size 500. We conclude that the retrieval results with SCOR are not affected by either the choice of the embeddings used or the sizes of the vectors involved.

RQ6: Does replacing DLM with TFIDF in MRF based retrieval enhance the performance of bug localization systems?

Since TFIDF performs better than FI in terms of retrieval precision, and is comparable in performance to the more advanced BoW tools like BugLocator and BLUiR as we discussed in the answer to RQ1, we believe tht the question posed above is important. The comparative results presented in Table 6 say that the answer to this question is a definite yes. That table shows the performance of MRF SD and SCOR using TFIDF as the BoW model versus the results shown previously in Table 4. The notation SD-T and SCOR-T is for these algorithms when use the TFIDF score for the BoW contribution when computing the composite relevance score of a file vis-a-vis a bug report.

Table 6: We compare MAP values of MRF SD with MRF SD-T, and SCOR with SCOR-T retrieval algorithms evaluated on Bugzbook dataset. Notice that SD-T and SCOR-T are the versions of MRF SD and SCOR when using TFIDF scores in computing the composite score for retrieval, respectively.

Project	SD	SD-T	SCOR	SCOR-T
Ambari	0.268	0.294	0.295	0.298
Aspectj	0.226	0.232	0.250	0.235
Bigtop	0.304	0.325	0.560	0.572
Camel	0.405	0.420	0.407	0.417
Cassandra	0.369	0.389	0.411	0.451
CXF	0.348	0.358	0.363	0.376
Drill	0.218	0.231	0.240	0.245
Eclipse	0.303	0.313	0.320	0.323
HBase	0.429	0.443	0.453	0.449
Hive	0.335	0.370	0.345	0.339
JCR	0.453	0.454	0.450	0.465
Karaf	0.374	0.393	0.427	0.424
Mahout	0.315	0.334	0.338	0.348
Math	0.545	0.519	0.512	0.481
Opennlp	0.433	0.470	0.498	0.510
PDFBox	0.368	0.381	0.380	0.394
PIG	0.295	0.353	0.335	0.396
SOLR	0.371	0.384	0.394	0.398
Spark	0.377	0.437	0.418	0.441
Sqoop	0.367	0.384	0.417	0.419
Tez	0.424	0.439	0.431	0.468
Tika	0.290	0.328	0.326	0.361
Wicket	0.450	0.458	0.440	0.440
WW	0.414	0.426	0.430	0.448
Zookeeper	0.565	0.507	0.529	0.524
Chrome	0.119	0.125	0.137	0.125
OpenCV	0.845	0.699	0.819	0.804
Pandas	0.375	0.365	0.435	0.437
Tensorflow	0.246	0.201	0.182	0.186
Average	**0.370**	**0.380**	**0.398**	**0.409**
MI-wtd	**0.467**	**0.490**	**0.500**	**0.512**

6 CONCLUSION

The roughly fifteen-year progression of research in IR-based search tools for source code libraries consists of three distinct phases: it started with tools based on the first-order statistical properties of the source-code files and the queries; then moved into augmenting the first-order properties with software-centric information derived from evolution history and structure; and, finally, into exploiting the proximity, order, as well as contextual semantics provided by the word2vec neural network. The tools developed along the way were tested on relatively small evaluation datasets. This paper provides a large-scale comprehensive evaluation with over 20,000 bug reports of a set of search tools that represent all the phases of this research. Our results consist of answers to six research questions that address the relative importance of the different components of the search logic. For future we intend to evaluate more retrieval algorithms from each generation on open-source as well industry projects.

REFERENCES

[1] S. Akbar and A. Kak. 2019. SCOR: Source Code Retrieval with Semantics and Order. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 1–12. https://doi.org/10.1109/MSR.2019.00012

[2] Shayan A. Akbar. [n.d.]. *SCOR Word Embeddings*. Retrieved March 2, 2020 from https://engineering.purdue.edu/RVL/SCOR_WordEmbeddings

[3] Apache. [n.d.]. *Apache Archives*. Retrieved March 2, 2020 from https://archive.apache.org

[4] Atlassian. [n.d.]. *Jira Issue Types*. Retrieved March 2, 2020 from https://confluence.atlassian.com/adminjiracloud/issue-types-844500742.html

[5] Atlassian. [n.d.]. *Jira Platform*. Retrieved March 2, 2020 from https://www.atlassian.com/software/jira

[6] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *TACL* 5 (2017), 135–146.

[7] Matej Cepl. [n.d.]. *GitHub Issues Export*. Retrieved March 2, 2020 from https://github.com/mcepl/github-issues-export

[8] Steven Davies and Marc Roper. 2013. Bug localisation through diverse sources of information. In *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 126–131.

[9] Steven Davies, Marc Roper, and Murray Wood. 2012. Using bug report similarity to enhance bug localisation. In *2012 19th Working Conference on Reverse Engineering*. IEEE, 125–134.

[10] X. Huo, F. Thung, M. Li, D. Lo, and S. Shi. 2019. Deep Transfer Bug Localization. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2920771

[11] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. [n.d.]. *BugLocator Software*. Retrieved March 2, 2020 from https://code.google.com/archive/p/bugcenter/downloads

[12] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. 2013. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering* 39, 11 (2013), 1597–1610.

[13] Adrian Kuhn, StÃĺphane Ducasse, and Tudor GÃőrba. 2007. Semantic clustering: Identifying topics in source code. *Information and Software Technology* 49, 3 (2007), 230 – 243. https://doi.org/10.1016/j.infsof.2006.10.017 12th Working Conference on Reverse Engineering.

[14] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. 2017. Bug Localization with Combination of Deep Learning and Information Retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. 218–229. https://doi.org/10.1109/ICPC.2017.24

[15] T. B. Le, M. Linares-Vasquez, D. Lo, and D. Poshyvanyk. 2015. RCLinker: Automated Linking of Issue Reports and Commits Leveraging Rich Contextual Information. In *2015 IEEE 23rd International Conference on Program Comprehension*. 36–47. https://doi.org/10.1109/ICPC.2015.13

[16] Jaekwon Lee, Dongsun Kim, Tegawendé F. Bissyandé, Woosung Jung, and Yves Le Traon. 2018. Bench4BL: Reproducibility Study of the Performance of IR-based Bug Localization. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. 1–12. https://doi.org/10.1145/3213846.3213856

[17] Stacy K. Lukins, Nicholas A. Kraft, and Letha H. Etzkorn. 2008. Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering (WCRE '08)*. IEEE Computer Society, Washington, DC, USA, 155–164. https://doi.org/10.1109/WCRE.2008.33

[18] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. 2004. An information retrieval approach to concept location in source code. In *11th Working Conference on Reverse Engineering*. 214–223. https://doi.org/10.1109/WCRE.2004.10

[19] Donald Metzler and W Bruce Croft. 2005. A Markov random field model for term dependencies. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 472–479.

[20] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 3111–3119. http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf

[21] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. 2014. On the use of stack traces to improve text retrieval-based bug localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 151–160.

[22] T. V. Nguyen, A. T. Nguyen, H. D. Phan, T. D. Nguyen, and T. N. Nguyen. 2017. Combining Word2Vec with Revised Vector Space Model for Better Code Retrieval. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 183–185. https://doi.org/10.1109/ICSE-C.2017.90

[23] Brent D. Nichols. 2010. Augmented Bug Localization Using Past Bug Information. In *Proceedings of the 48th Annual Southeast Regional Conference (ACM SE âĂŹ10)*. Association for Computing Machinery, New York, NY, USA, Article Article 61, 6 pages. https://doi.org/10.1145/1900008.1900090

[24] University of Glasgow. [n.d.]. *Terrier Open-source Search Engine Software*. Retrieved March 2, 2020 from http://terrier.org/

[25] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *In EMNLP*.

[26] Mohammad Masudur Rahman and Chanchal K Roy. 2018. Improving ir-based bug localization with context-aware query reformulation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 621–632.

[27] Shivani Rao and Avinash Kak. 2011. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 43–52.

[28] Stephen E Robertson and Karen Spärck Jones. 1994. *Simple, proven approaches to text retrieval*. Technical Report. University of Cambridge, Computer Laboratory.

[29] Ripon K. Saha. [n.d.]. *BLUiR Software*. Retrieved December 2, 2019 from http://riponsaha.com/BLUiR.html

[30] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 345–355.

[31] Bunyamin Sisman, Shayan A Akbar, and Avinash C Kak. 2017. Exploiting spatial code proximity and order for improved source code retrieval for bug localization. *Journal of Software: Evolution and Process* 29, 1 (2017).

[32] Bunyamin Sisman and Avinash C. Kak. [n.d.]. *BUGLinks Dataset*. Retrieved March 2, 2020 from https://engineering.purdue.edu/RVL/Database/BUGLinks/

[33] Bunyamin Sisman and Avinash C Kak. 2012. Incorporating version histories in information retrieval based bug localization. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE, 50–59.

[34] Bunyamin Sisman and Avinash C Kak. 2013. Assisting code search with automatic query reformulation for bug localization. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 309–318.

[35] Stephen W Thomas, Meiyappan Nagappan, Dorothea Blostein, and Ahmed E Hassan. 2013. The impact of classifier configuration and classifier combination on bug localization. *IEEE Transactions on Software Engineering* 39, 10 (2013), 1427–1443.

[36] Y. Uneno, O. Mizuno, and E. Choi. 2016. Using a Distributed Representation of Words in Localizing Relevant Files for Bug Reports. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 183–190. https://doi.org/10.1109/QRS.2016.30

[37] Thomas Zimmermann Valentin Dallmeier. [n.d.]. *iBUGS Dataset*. Retrieved March 2, 2020 from https://www.st.cs.uni-saarland.de/ibugs/

[38] Shaowei Wang and David Lo. 2014. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 53–63.

[39] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 262–273.

[40] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 181–190.

[41] Yan Xiao, Jacky Keung, Kwabena E. Bennin, and Qing Mi. 2019. Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology* 105 (2019), 17 – 29. https://doi.org/10.1016/j.infsof.2018.08.002

[42] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. Learning to Rank Relevant Files for Bug Reports Using Domain Knowledge. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 689–699. https://doi.org/10.1145/2635868.2635874

[43] Xin Ye, Razvan Bunescu, and Chang Liu. 2016. Mapping Bug Reports to Relevant Files: A Ranking Model, a Fine-Grained Benchmark, and Feature Evaluation. *IEEE Trans. Softw. Eng.* 42, 4 (April 2016), 379âĂŞ402. https://doi.org/10.1109/TSE.2015.2479232

[44] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu. 2016. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 404–415. https://doi.org/10.1145/2884781.2884862

[45] K. C. Youm, J. Ahn, J. Kim, and E. Lee. 2015. Bug Localization Based on Code Change Histories and Bug Reports. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*. 190–197. https://doi.org/10.1109/APSEC.2015.23

[46] Chengxiang Zhai and John Lafferty. 2017. A study of smoothing methods for language models applied to ad hoc information retrieval. In *ACM SIGIR Forum*, Vol. 51. ACM, 268–276.

[47] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 14–24.