Where Should the Bugs Be Fixed?

More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports

Presented by: Chandani Shrestha
For CS 6704 class

About the Paper and the Authors

Publication date: 2012/6/2
Proceedings of the 34th International Conference on Software Engineering
IEEE press
Cited by 181

Jian Zhou - Professor of Mathematics, Tsinghua University, China
Visiting Assistant Professor(1995 - 2001): University of California at Santa Barbara, Texas A&M University, Massachusetts Institute of Technology

Hongyu Zhang - Associate Professor, University of Newcastle, Australia
1999 - 2016: Lead Researcher at Microsoft Research, Associate professor at Tsinghua University, Lecturer at RMIT University, Australia, Software engineer at IBM Singapore

David Lo - Associate Professor, Singapore Management University, Singapore
2014: Visiting Researcher Microsoft Research

Bug Localization

Bug Localization

- Despite software quality assurance activities (testing, inspection, static checking), software system comes with defects (bugs)
- Locating the source code files that contains the bugs reported

Need for Automated Bug Localization

- When number of files and reports large, manually locating bugs is time consuming time-bug fix time is prolonged, maintenance cost, customer dissatisfaction

Information Retrieval-Based Bug Localization

Treat bug report and the source code files as text documents $\longrightarrow$ Compute textual similarity of source code files with a given bug report $\longrightarrow$ Present the ranked relevant files based on the similarity $\longrightarrow$ Developers investigate the files till the relevant buggy file is found

A common bug localization process, which consists of four steps:

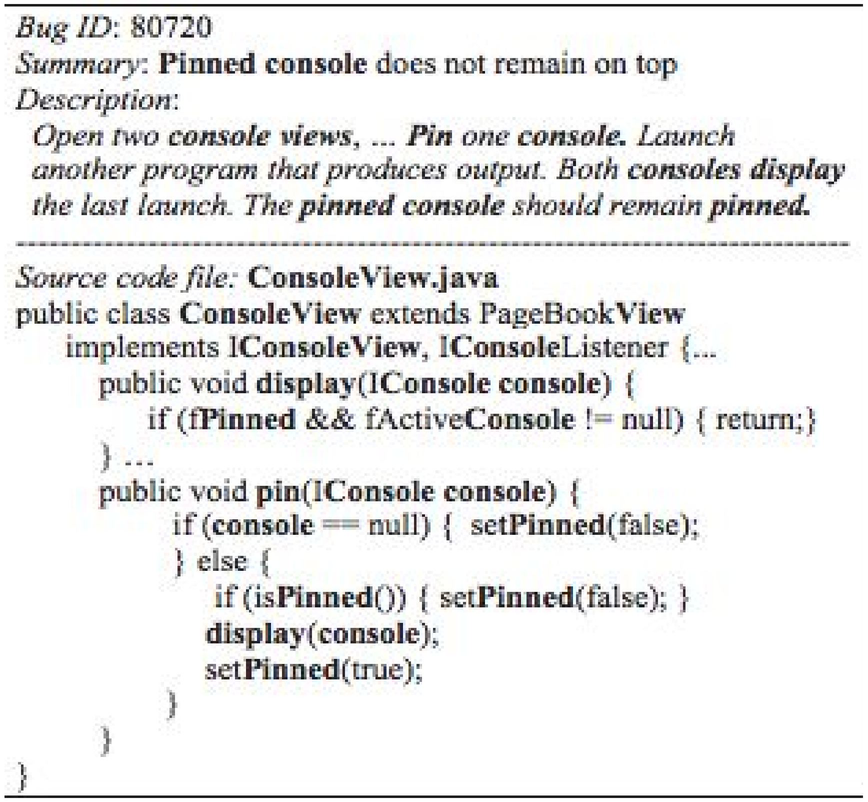
Corpus creation: This step performs lexical analysis for each source code file and creates a vector of lexical tokens. (key word, stop-word removal, stemming)

Indexing: Documents are indexed for fast retrieval.

Query construction: Bug report as a query - uses extracted tokens to search for relevant files in the indexed source code corpus

Retrieval & ranking: Based on the textual similarity between the query and each of the files in the corpus.

Example



Bug ID: 80720
Summary: **Pinned console** does not remain on top
Description:
*Open two **console views**, ... **Pin** one **console**. Launch another program that produces output. Both **consoles display** the last launch. The **pinned console** should remain **pinned**.*

Source code file: **ConsoleView.java**

```
public class ConsoleView extends PageBookView
    implements IConsoleView, IConsoleListener {...
      public void display(IConsole console) {
          if (fPinned && fActiveConsole != null) { return;}
      } ...
      public void pin(IConsole console) {
           if (console == null) {  setPinned(false);
           } else {
             if (isPinned()) { setPinned(false); }
             display(console);
             setPinned(true);
           }
      }
}
```

Figure 1. A bug report and its relevant source code file

Bug report (ID: 80720) for Eclipse 3.1.

The bug report (including bug summary and description) contains many words such as pin(pinned), console, view, display, etc.

In Eclipse 3.1, there is a source code file called ConsoleView.java, which also contains many occurrences of the similar words.

Treat the bug report and the source code files as text documents, and compute the textual similarity between them.

For a corpus of files, rank the files based on each file's textual similarity to the bug report.

The goal of bug localization is to rank the buggy files as high as possible in the list (Various approaches can be used to compute a relevance score)

Information Retrieval Approaches

- Smoothed Unigram Model (SUM) - best performing model
- Latent Dirichlet Allocation (LDA)
- Latent Semantic Indexing (LSI)
- Vector Space Model (VSM)
    - Queries and documents are represented as vectors of weights, where each weight corresponds to a term.
    - The value of each weight is usually the term frequency—inverse document frequency (TF-IDF) of the corresponding word.
        - Term frequency: number of times a word appears in a document.
        - Inverse document frequency : number of documents that contain the word.
    - The higher the term frequency and inverse document frequency of a word, the more important the word would be.

Limitations:

Low accuracy: The performance of existing bug localization methods can be further improved. **Example:** Using LDA, relevant files of only 22% Eclipse 3.1 bugs are ranked in the top 10.

Small-scale experiments: Used a small number of selected bug reports in their evaluation. **Example:** Method called PROMESIR, utilized Latent Semantic Indexing to locate 3 bugs in Eclipse and 5 bugs in Mozilla.

Proposed approach: BugLocator

- **Uses revised Vector Space Model (rVSM) to rank all source code files based on an initial bug report**
  - take the document length into consideration optimize the classic VSM model for bug localization.
- **Adjust the obtained ranks by using information of the similar bugs that have been fixed before.**

Aspects considered:

- **Source code files:** May contain words that are similar to those occurring in the bug reports. Analyzing source code files can help determine the location where the bug has impact on.
- **Similar bugs:** Once a new bug report is received, we can examine similar bugs that were reported and fixed before. The information on locations where past similar bugs were fixed could help us locate the relevant files for the new bug.
- **Software size:** When two source files have similar scores, we need to determine which one should be ranked higher. From previous works - statistically, larger files are more likely to contain bugs. Therefore for bug localization we need to assign higher scores to larger files.

Overall Structure of BugLocator

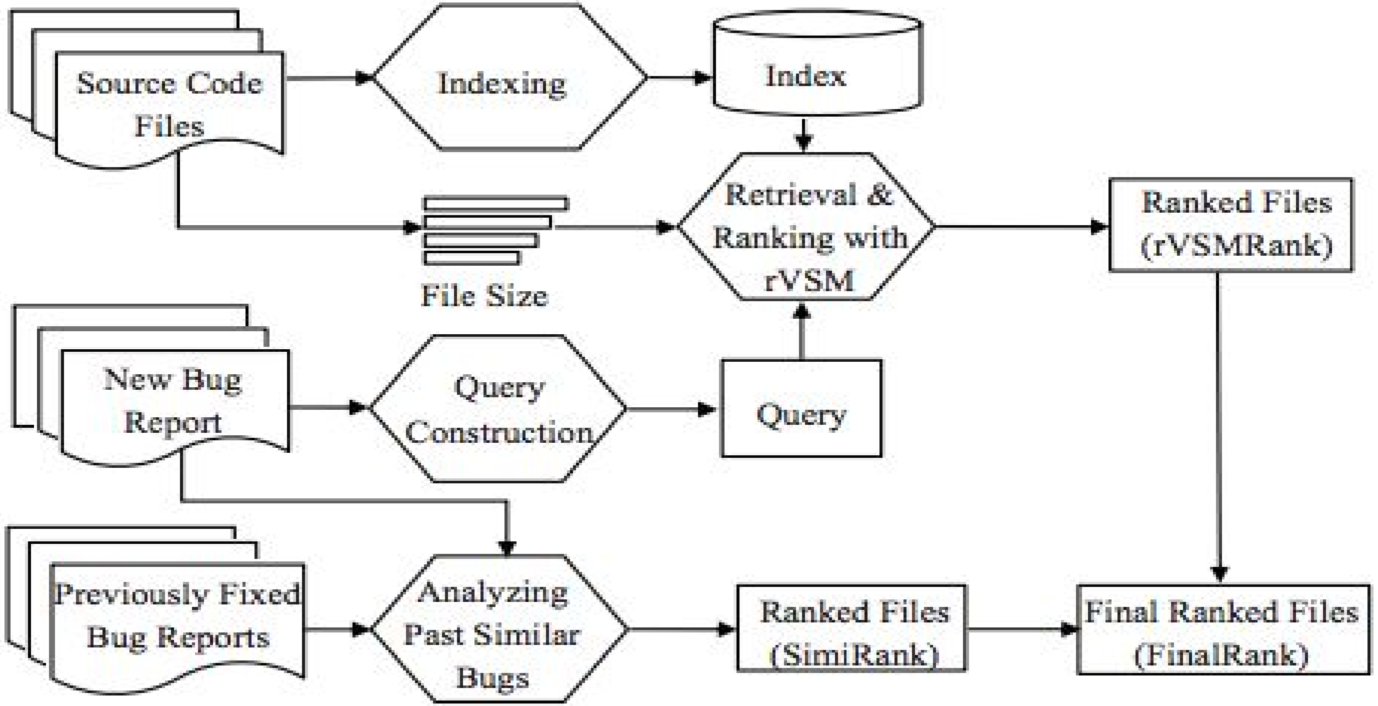


Figure 2. The overall structure of BugLocator

Ranking based on Source Code Files

- The similarity between each file and the bug report is computed.
- The files are then ranked by the similarity values and returned as output.

In a classic VSM, the relevance score between a document d and a query q is computed as the cosine similarity between their corresponding vector representations:

$$Similarity(q,d) = \cos(q,d) = \frac{\overline{V_q} \bullet \overline{V_d}}{|\overline{V_q}||\overline{V_d}|}$$

The term weight w is computed based on the term frequency (tf) and the inverse document frequency (idf). In classic VSM, tf and idf are defined as follows:

$$tf(t,d) = \frac{f_{td}}{\#terms}, idf(t) = \log(\frac{\#docs}{n_t})$$

Many variants of tf(t,d) have been proposed to improve the performance of the VSM including logarithm, augmented, and Boolean variants of the classic VSM. It is observed that the logarithm variant can lead to better performance

$$tf(t,d) = \log(f_{td}) + 1$$

Continue...

- Classical VSM favours small documents during ranking as long documents are often poorly represented because they have poor similarity values.
- Larger source code files tend to have higher probability of containing a bug, so need to rank larger files higher. We thus define a function g to model the document length in rVSM:

$$g(\#terms) = \frac{1}{1 + e^{-N(\#terms)}}$$

(Used to compute the length value for each source file according to the number of terms the file contains.)

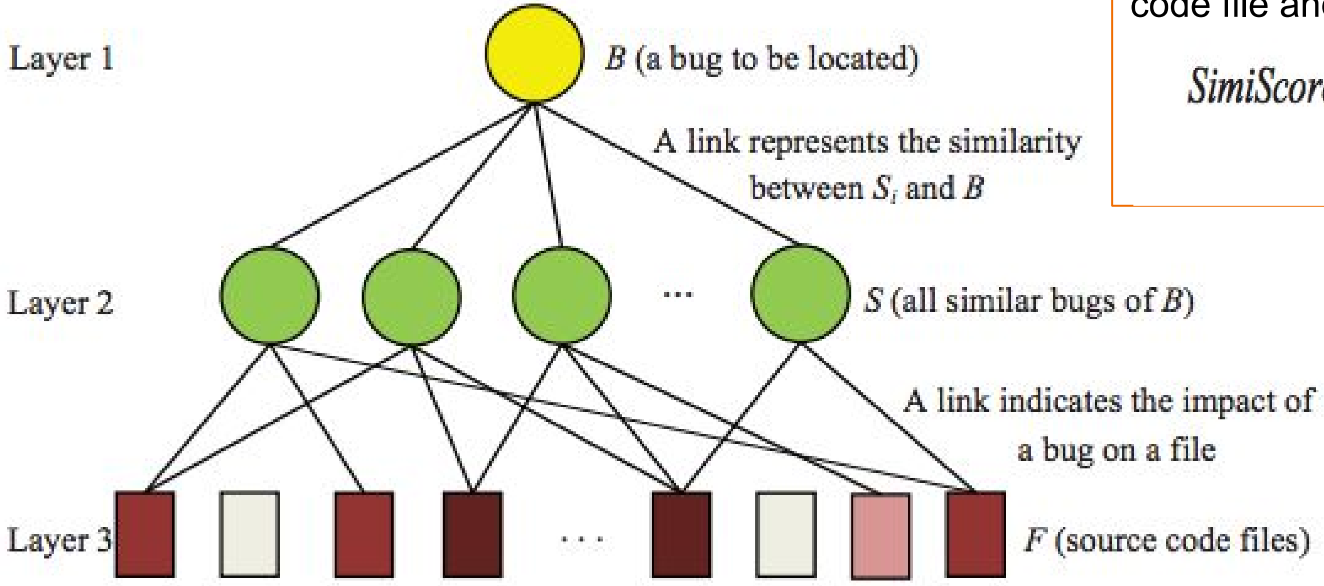
New scoring algorithm for rVSM:

$$rVSMScore(q,d) = g(\#term) \times \cos(q,d)$$

Ranking Based on Similar Bugs

- Examine similar bugs that have been fixed before - adjust the rankings of the relevant files.
- Assumption: Similar bugs tend to fix similar files.

Proposed method:



Layer 1
B (a bug to be located)
A link represents the similarity between S_i and B
Layer 2
S (all similar bugs of B)
A link indicates the impact of a bug on a file
Layer 3
F (source code files)

Figure 3. Heterogeneous Bug-File graph

Degree of relevance between a source code file and the bug B:

$$SimiScore = \sum_{\substack{\text{All } S_i \text{ that} \\ \text{connect to } F_j}} (Similarity(B, S_i) / n_i)$$

Combining Ranks

Combining the VSMScore and SimiScore:

$$FinalScore = (1-\alpha)\times N(rVSMScore) + \alpha\times N(SimiScore)$$

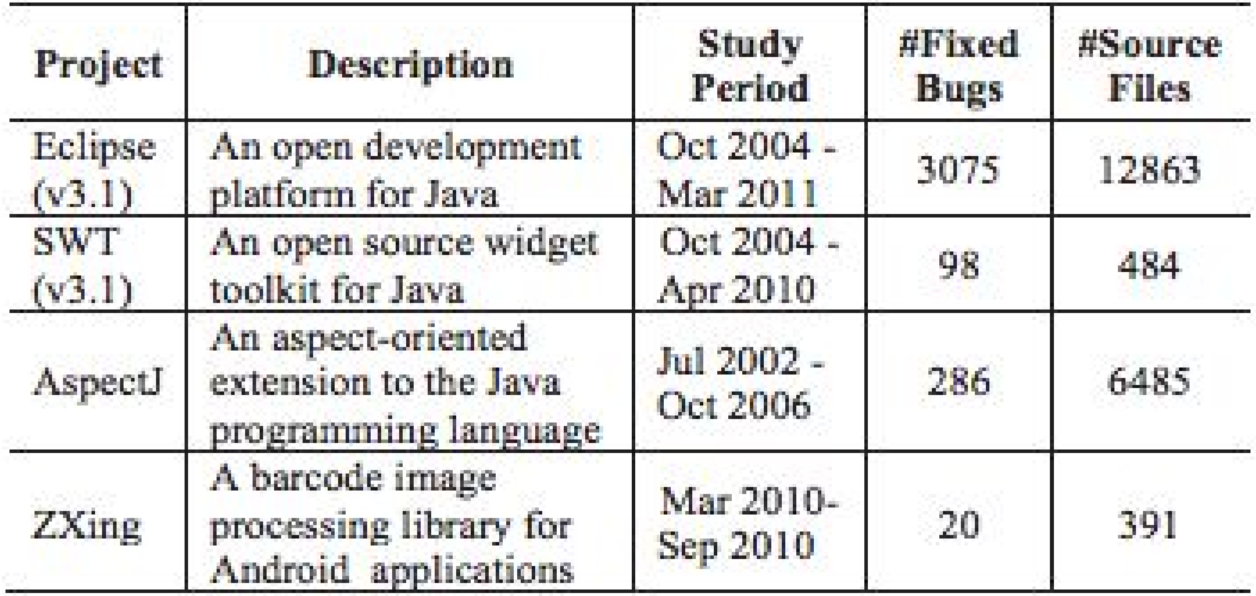
- ***α*** is a weighting factor and valued between 0-1(inclusive)
- It adjusts the weights of the two rankings.
- Value (***α***) can be set empirically- when ***α*** is between 0.2 and 0.3, the proposed method performs the best.

Source code files ranked by FinalScore in descending order are returned to users (FinalRank).

Experimental Setup

Evaluated Projects:

TABLE I. THE STUDIED PROJECTS



Project	Description	Study Period	#Fixed Bugs	#Source Files
Eclipse (v3.1)	An open development platform for Java	Oct 2004 - Mar 2011	3075	12863
SWT (v3.1)	An open source widget toolkit for Java	Oct 2004 - Apr 2010	98	484
AspectJ	An aspect-oriented extension to the Java programming language	Jul 2002 - Oct 2006	286	6485
ZXing	A barcode image processing library for Android applications	Mar 2010- Sep 2010	20	391

Data Collection

- For each subject system, collected initial bug reports from the bug tracking system (such as BugZilla).
- To evaluate the bug localization performance, only the bug reports of fixed bugs collected.
- To establish the links between bug reports and source code files, we adopt the traditional heuristics proposed by Bachmann and Bernstein:
    - Scan through the change logs for bug IDs in a given format (e.g. “issue 681”, “bug 239” and so on).
    - Exclude all false-positive bug numbers (e.g. “r420”, “2009-05-07 10:47:39 -0400” and so on).
    - Check if there are other potential bug number formats or false positive number formats, add the new formats and scan the change logs iteratively.
    - Check if potential bug numbers exist in the bugtracking database with their status marked as fixed.

Research Questions

RQ1: How many bugs can be successfully located by BugLocator?

- For each bug report, obtain the relevant files that have been modified to fix the bug.
- Check the ranks of these files in the query results returned by BugLocator.
- If the files are ranked in top 1, top 5 or top 10, we consider the report has been effectively localized.
- Perform the experiment for all bug reports and calculate the percentage of bugs that have been successfully located.
- Compute the Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR) measures

Research Questions Continued...

RQ2: Does the revised Vector Space Model (rVSM) improve the bug localization performance?

- To evaluate the effectiveness of rVSM, we perform bug localization on the subject systems using classic and revised VSM, and compare the results.

Research Questions Continued...

RQ3: Does the consideration of similar bugs improve the bug localization performance?

- Perform bug localization on the four subject systems with/without the rankings learned from past similar bugs.
- The parameter α adjusts the weights of the two rankings. When $\alpha = 0$, the final rank is only dependent on the queries of source code files. When the value of α is between 0 and 1, the final rank is a combination of two ranking results.
- Also evaluate the effect of different α values.

$$FinalScore = (1-\alpha) \times N(rVSMScore) + \alpha \times N(SimiScore)$$

Research Questions Continued...

RQ4: Can BugLocator outperform other bug localization methods?

- Compare BugLocator to the bug localization methods implemented using the following IR techniques:
    - Smoothed Unigram Model (SUM) - best performing model
    - Latent Dirichlet Allocation (LDA)
    - Latent Semantic Indexing (LSI)
    - Vector Space Model (VSM)

Evaluation Metrics

Top N Rank:

- The number of bugs whose associated files are ranked in the top N (N= 1, 5, 10) of the returned results.
- If the top N query results contain at least one file at which the bug should be fixed, we consider the bug located.

MRR (Mean Reciprocal Rank):

- The reciprocal rank of a query is the multiplicative inverse of the rank of the first correct answer.
- The mean reciprocal rank is the average of the reciprocal ranks of results of a set of queries Q.

MAP (Mean Average Precision):

- Provides a single-figure measure of quality of information retrieval, when a query may have multiple relevant documents.
- MAP for a set of queries is the mean of the average precision values for all queries.

Experimental Results

RQ1: How many bugs can be successfully located by BugLocator?

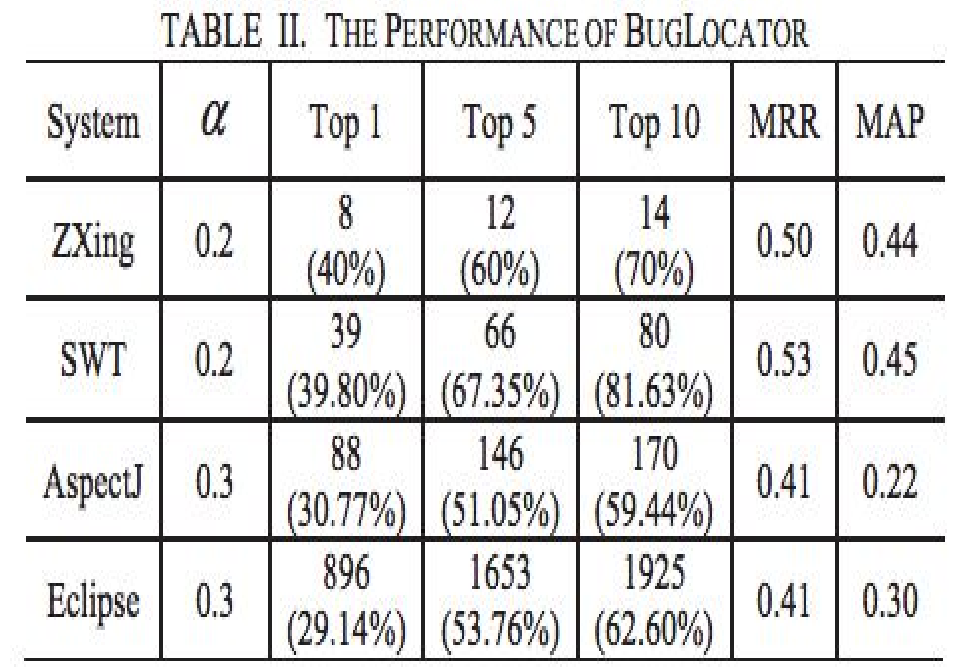


TABLE II. THE PERFORMANCE OF BUGLOCATOR

System	α	Top 1	Top 5	Top 10	MRR	MAP
ZXing	0.2	8 (40%)	12 (60%)	14 (70%)	0.50	0.44
SWT	0.2	39 (39.80%)	66 (67.35%)	80 (81.63%)	0.53	0.45
AspectJ	0.3	88 (30.77%)	146 (51.05%)	170 (59.44%)	0.41	0.22
Eclipse	0.3	896 (29.14%)	1653 (53.76%)	1925 (62.60%)	0.41	0.30

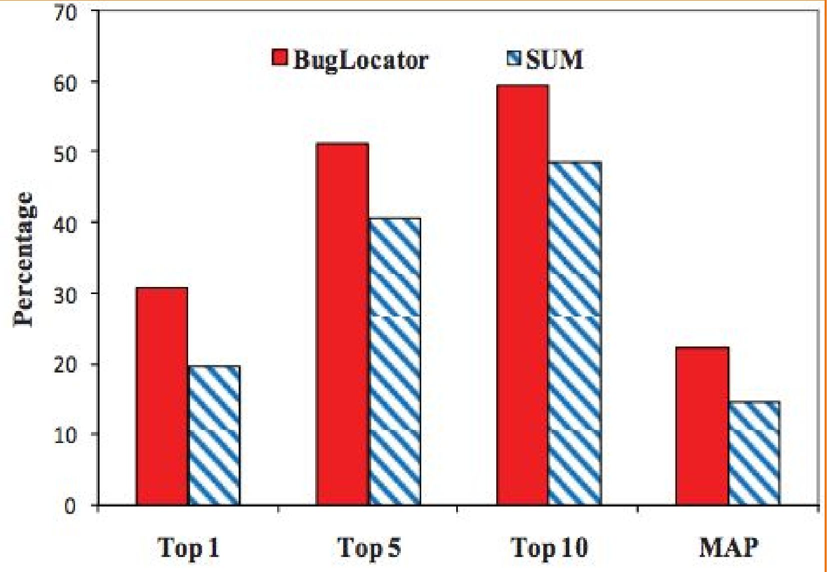


Figure 4. The comparison between the results of BugLocator and the SUM results given in [32] on AspectJ dataset

Experimental Results Continued

RQ2: Does the revised Vector Space Model (rVSM) improve the bug localization performance?

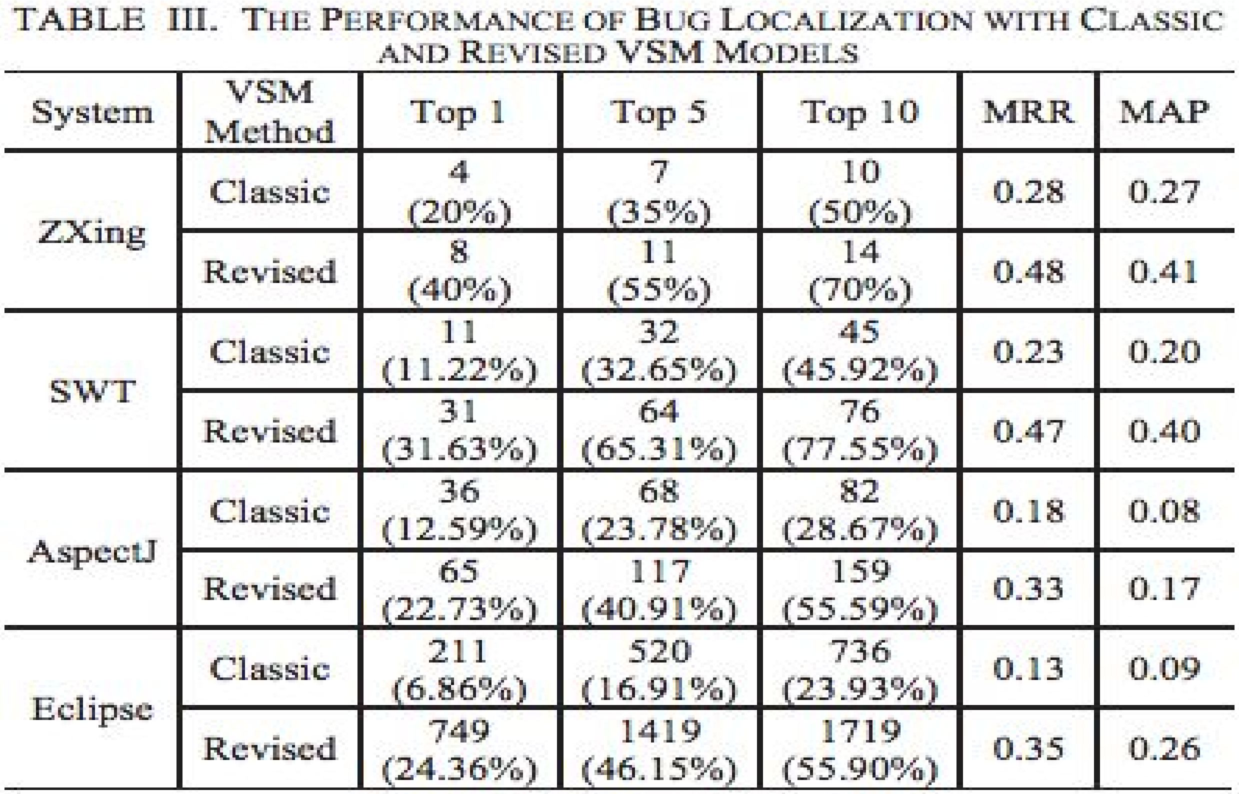


TABLE III. THE PERFORMANCE OF BUG LOCALIZATION WITH CLASSIC AND REVISED VSM MODELS

System	VSM Method	Top 1	Top 5	Top 10	MRR	MAP
ZXing	Classic	4 (20%)	7 (35%)	10 (50%)	0.28	0.27
	Revised	8 (40%)	11 (55%)	14 (70%)	0.48	0.41
SWT	Classic	11 (11.22%)	32 (32.65%)	45 (45.92%)	0.23	0.20
	Revised	31 (31.63%)	64 (65.31%)	76 (77.55%)	0.47	0.40
AspectJ	Classic	36 (12.59%)	68 (23.78%)	82 (28.67%)	0.18	0.08
	Revised	65 (22.73%)	117 (40.91%)	159 (55.59%)	0.33	0.17
Eclipse	Classic	211 (6.86%)	520 (16.91%)	736 (23.93%)	0.13	0.09
	Revised	749 (24.36%)	1419 (46.15%)	1719 (55.90%)	0.35	0.26

Experimental Results Continued

RQ3: Does the consideration of similar bugs improve the bug localization performance?

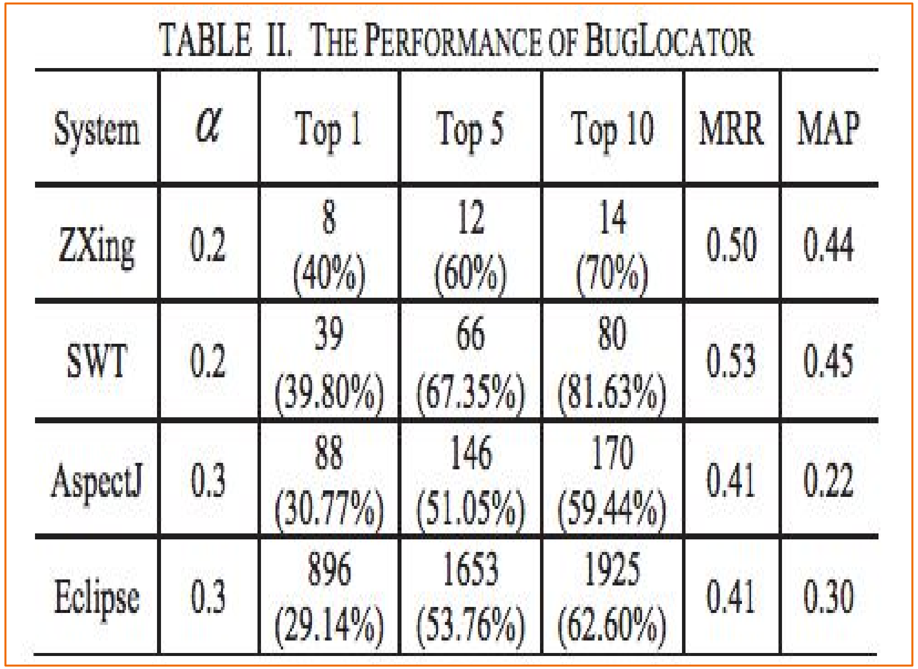


TABLE II. THE PERFORMANCE OF BUGLOCATOR

System	α	Top 1	Top 5	Top 10	MRR	MAP
ZXing	0.2	8 (40%)	12 (60%)	14 (70%)	0.50	0.44
SWT	0.2	39 (39.80%)	66 (67.35%)	80 (81.63%)	0.53	0.45
AspectJ	0.3	88 (30.77%)	146 (51.05%)	170 (59.44%)	0.41	0.22
Eclipse	0.3	896 (29.14%)	1653 (53.76%)	1925 (62.60%)	0.41	0.30

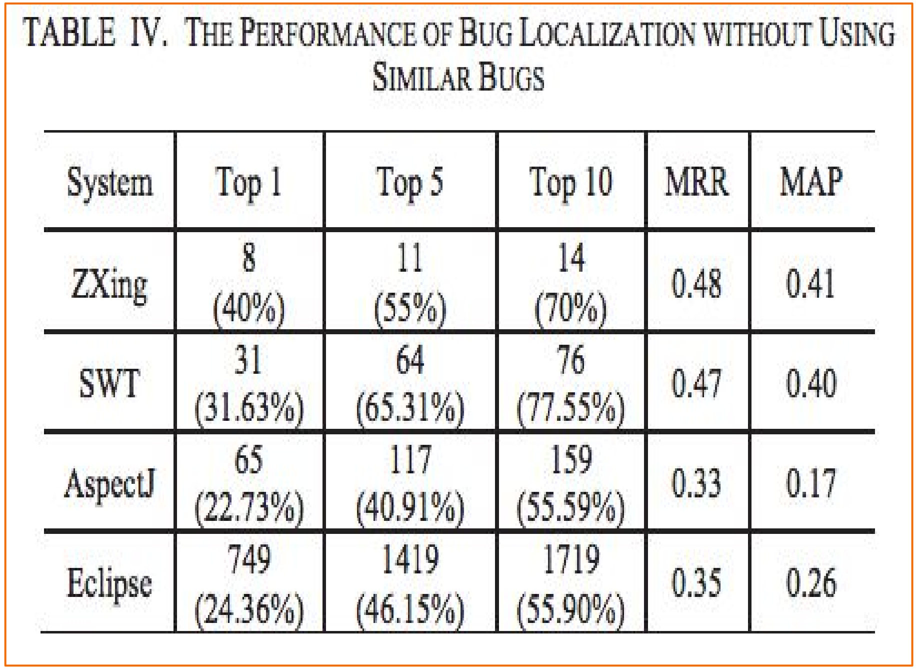


TABLE IV. THE PERFORMANCE OF BUG LOCALIZATION WITHOUT USING SIMILAR BUGS

System	Top 1	Top 5	Top 10	MRR	MAP
ZXing	8 (40%)	11 (55%)	14 (70%)	0.48	0.41
SWT	31 (31.63%)	64 (65.31%)	76 (77.55%)	0.47	0.40
AspectJ	65 (22.73%)	117 (40.91%)	159 (55.59%)	0.33	0.17
Eclipse	749 (24.36%)	1419 (46.15%)	1719 (55.90%)	0.35	0.26

Experimental Results Continued for RQ3

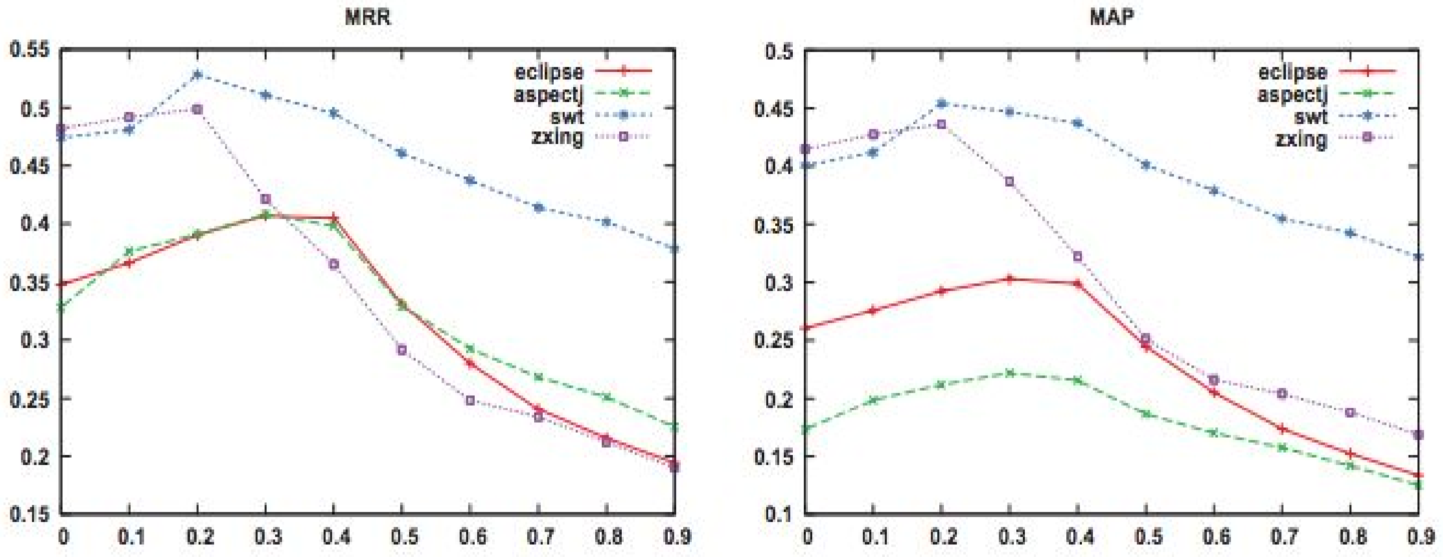


Figure 5. The impact of α on bug localization performance (MAP and MRR)

Experimental Results Continued

RQ4: Can BugLocator outperform other bug localization methods?

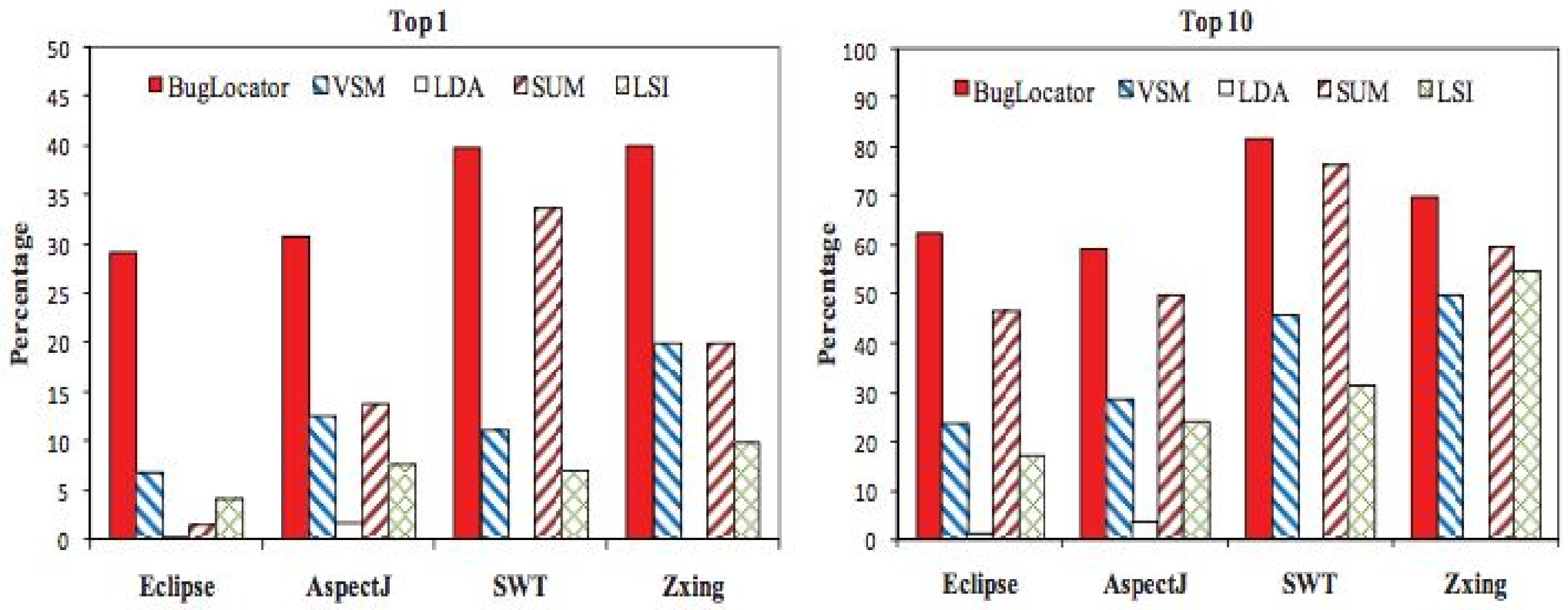


Figure 6. The comparisons between different bug localization methods

Threats to Validity

- The nature of the data in open source projects may be different from those in projects developed by well-managed software organizations.
- A limitation of our approach is that we rely on good programming practices in naming variables, methods and classes.
- Our approach also relies on the quality of bug reports. If a bug report does not provide enough information, or provides misleading information, the performance of BugLocator is adversely affected.

Discussion

1. Why does the proposed rVSM method work?
2. Why can similar bugs help improve bug localization performance? (what context would it be more useful)

All evaluated project Java based, would there be any significant change in performance if IR techniques used other language projects?
Problem with such large scale experiment?

Why does the proposed rVSM method work?

- It is often not the case that the term importance is proportional to its occurrence frequency, so high frequency may have negative impact on information retrieval performance.
- The logarithm variant of tf can help smooth the impact of the high frequent terms
- Larger files tend to be more defect-prone than the smaller files- uses a logistic function g to adjust the ranking results

Why can similar bugs help improve bug localization performance?

- We find out that for many bugs, the associated files have overlaps with the associated files of their similar bugs.
  Context
- The analysis of similar bugs becomes more important when the textual similarity between bug reports and source code is low.
- For the Eclipse **bug 89014** was fixed in the file BindingComparator.java. Using rVSM, the relevant file BindingComparator.java is only **ranked 2527**, because the textual similarity between source code and the bug report is low. Aanalysis on similar bugs found that it is similar to previous fixed **bugs 83817**, **79609** and **79544**, all fixed in BindingComparator.java. BugLocator combines the scores obtained from rSVM and similar bug, and the final rank of the file BindingComparator.java becomes **7**.

All evaluated project Java based, would there be any significant change in performance if IR techniques used other language projects?

“We noted previously that the Java projects use a substantially higher rate of English words than the C projects. Nevertheless, particularly in terms of Recall at Top 1, MAP, and MRR, we get equal or better results for the C projects than for the Java projects.”

R. K. Saha, J. Lawall, S. Khurshid, D. E. Perry, "On the effectiveness of information retrieval based bug localization for c programs", *ICSME*, pp. 161-170, 2014.

Problem with such large scale experiment?

Bugs that were previously fixed are no longer present in that code, and for old bug reports, the code may have changed substantially since the bug was encountered. Ideally, for each bug report we would extract the version from when the bug was reported to get the actual buggy code. However, this approach is impractical for a large-scale experiment.

R. K. Saha, J. Lawall, S. Khurshid, D. E. Perry, "On the effectiveness of information retrieval based bug localization for c programs", *ICSME*, pp. 161-170, 2014.