Applied Soft Computing 62 (2018) 636–648

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

Full length article

Comparing learning to rank techniques in hybrid bug localization

Zhendong Shi[a,b], Jacky Keung[b], Kwabena Ebo Bennin[b], Xingjun Zhang[a,*]

[a] Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China
[b] Department of Computer Science, City University of Hong Kong, Hong Kong, China

ARTICLE INFO

Article history:
Received 9 May 2017
Received in revised form 21 October 2017
Accepted 29 October 2017
Available online 8 November 2017

Keywords:
Bug localization
Information retrieval
Learn to rank
Fault localization

ABSTRACT

Bug localization is a software development and maintenance activity that aims to find relevant source code entities to be modified so that a specific bug can be fixed on the basis of the given bug report. Information retrieval (IR) techniques have been widely used to locate bugs in recent decades. These techniques mainly use the IR similarity between the bug report and source code entities. In addition to IR similarity, features that are extracted from version history, source code structure, dynamic analysis, and other resources are found to be beneficial for bug localization. The approaches utilizing extra features in IR-based bug localization are called hybrid bug localization. We conduct a short survey of the hybrid bug localization methods that use additional features in addition to IR similarity. We also use Learning to Rank (LtR) techniques to combine the beneficial features to improve bug localization. Learning to Rank is the application of machine learning in the ranking models for information retrieval. We compared eight LtR techniques in bug localization, and the experimental results show that coordinate ascent algorithms without normalization is a suitable LtR technique in bug localization for selected attributes, and it outperforms two state-of-the-art localization approaches for two large projects, Eclipse and SWT.

1. Introduction

A software project in development or maintenance phases most likely contains bugs. Bug localization is the process of using automatic or semi-automatic techniques to find the buggy locations in a source code. Bug localization has elicited considerable attention in recent decades. As developers usually name the classes/methods/variables after their functionalities and write comments with relevant terms, the submitted bug reports typically include terms related to buggy functionalities. The bug reports are therefore likely to have similar textual terms with relevant source-code entities. Information retrieval (IR), the main technique used in bug localization, refers to the process of finding documents that satisfy an information need from a large collection of documents [33]. Many IR-based bug localization techniques have been studied in recent years [32,43,45,50,52,56,58].

In addition to mainly depending on IR similarities, many hybrid approaches that use additional features from source code structure [2,47,50], dynamic analysis [20,30,42], version history [51,58], and others have been proposed in the literature. These hybrid approaches use additional features in favor of bug localization, and therefore perform significantly better than the approaches that use only IR similarities. A survey of these hybrid methods and their corresponding features is presented in this paper. However, many features are linearly combined with preset weight values [45,52,58,59,67], thereby indicating that the linear weight values are not learned from the data.

Combining the found beneficial features in a suitable manner may improve the performance of bug localization. Learning to Rank (LtR) is a set of machine learning-based ranking algorithms in the information retrieval domain. It has been widely used in web searching area in the past 10 years [31], provides an automated framework for combining different features in ranking task [6]. Ye et al. [62,63] have used LtR techniques in bug localization, but they only used one LtR algorithm.

In this paper, we compare the performance of eight LtR techniques in bug localization; such an approach combines the features that have been used in other studies. The features are selected from previous hybrid bug localization studies, and the feature weight values in LtR techniques are learned from historical bug data and source code information. When more beneficial features are found in future, they could be imported into this approach to increase the localization precision. The results show that the coordinate ascent algorithm [34], which learns weights in a linear combination of

* Corresponding author at: Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China.
E-mail addresses: zdshi0@stu.xjtu.edu.cn (Z. Shi), jacky.keung@cityu.edu.hk (J. Keung), kebennin2-c@my.cityu.edu.hk (K.E. Bennin), xjzhang@mail.xjtu.edu.cn (X. Zhang).

https://doi.org/10.1016/j.asoc.2017.10.048
1568-4946/

Bug ID	120704
Open date	2005-12-13 15:14:00
Fix date	2006-08-09 13:05:00
Summary	[Wizards] Problems with the **"New" wizard**
Description	1. There is a small red square in the top right. 2. The filter does not line up on the right with the viewer and looks bad. 3. The viewer is small vertically, and yet there is a lot of real estate that is being unused on the bottom. 4. Why is there a help icon on the bottom right, and why is it never enabled? Screenshot to follow.
Fixed files	org.eclipse.ui.internal.dialogs.**NewWizardNew**Page.java

Fig. 1. Bug #120704 of Eclipse project.

features without normalization, is a suitable means for bug localization in our selected features and projects, and it outperforms two state-of-art localization approaches for two large projects, namely, Eclipse and SWT.

The rest of this paper is structured as follows. We describe the background of hybrid bug localization, LtR techniques, and our LtR-based hybrid approach in Section 2. The survey of hybrid methods is presented in Section 3. We introduce the features selected in the LtR-based approach and show the results of applying LtR techniques in bug localization in Section 4. The threats to validity are discussed in Section 5. The conclusion is presented in Section 6.

2. Background

2.1. Bug localization approaches

2.1.1. Information retrieval based bug localization

To find the location of buggy parts in the source code, static and dynamic methods are used in bug localization. In static methods, a bug report is taken as input, and a ranked list of potential buggy source entities (files/methods) are returned. A bug report is written and submitted by software users, which describes the bug they encounter. An example of a bug report in the Eclipse project is shown in Fig. 1, which can be downloaded from Eclipse's Bugzilla.[1] A bug report usually contains a Bug ID, open date (the date a bug was submitted), fix date (the date a bug was fixed), summary, and description (detailed information of a bug). Fixed files are extracted from the version control system by searching text such as "Fixed 120704" or "bug #120704" in log messages [14]. Information retrieval techniques use the summary and description text of a bug report as the query and then search for textual similar source code files. Similar to web search engines [4], bug reports are considered input queries and source code files are considered candidate web pages.

In conventional IR-based bug localization, the documents (source code files) and the queries (bug reports) are preprocessed by Natural Language Processing (NLP), which includes splitting multi-word terms, removal of stop words and stemming [35]. For example, the camel case based term "*NewWizardNewPage*" is split into three words: "*new*", "*wizard*", and "*page*". The stop words which need to be removed from document collection include a list of meaningless words, such as "*to*", "*the*", "*do*", and program language keywords like "*for*"; "*while*"; etc. The stemmer like Porter [40] is to derive the root words from terms in documents; which transforms the words like "*opening*"; "*opens*"; and "*opened*" into their common root word "*open*". Consequently; the same terms in different tenses and variants can be recognized as the same.

After the text preprocessing, the documents and queries are represented in suitable IR models such as VSM (Vector Space Model) [46], SUM (Smoothed Unigram Model) [65], LSI (Latent Semantic Indexing) [17] and LDA (Latent Dirichlet Allocation) [7]. A brief description of these representation models are presented below:

VSM: A document or a query is represented as a term weighted vector usually using tf-idf (term frequency, inverse document frequency) scheme.

SUM: Unigram model is a probabilistic model in which the documents are ranked based on the probabilities to produce all terms of the query for each document. The model is smoothed to avoid zero probability, which is usually called Smoothed Unigram Model (SUM).

LSI: LSI uses singular value decomposition (SVD) to produce a low-dimensional space to represent documents from term-document matrixes. LSI is also called LSA (Latent Semantic Analysis).

LDA: LDA is a probabilistic topic model which introduces a topic layer between the document and words. Each topic is represented as a vector of term features, and each document is represented as a vector of the topic feature.

Apart from SUM which provides the ranking scheme in the model, the other three models only provide a way to represent documents. The ranked documents list is produced by applying similarity measures such as the cosine similarity [33]. For example, the fixed file name and the bug report summary in Fig. 1 share the common root terms, "new" and "wizard", so the text similarity between the fixed file and the bug report will be high. Therefore, the fixed file will be ranked highly in the final list.

Information retrieval is the main technique that is used in static analyzing methods to localize the bug, whereas dynamic methods [1,24], which are also called spectrum-based bug/fault localization, mainly examine the pass-and-fail execution traces of programs to provide suspiciousness scores to each line of source code. Dynamic methods involve running test cases to obtain the necessary information. In this paper, we only focus on hybrid bug localization approaches based on Information Retrieval and the approaches combining IR-based and spectrum-based techniques.

2.1.2. Bug localization vs. fault localization vs. feature location

Feature location usually refers to the process of finding the location in source code that corresponds to specific functionalities (features) [19]. Some researchers treat bug localization as a subarea of feature location [16,19]; feature location can be called *bug localization* when all features are bugs, and bugs are treated as unwanted features of software systems. Some other researchers [45,58] consider feature location as a different area from bug localization and think that features do not contain bugs. In their view, features only contain the feature or concept that developers want to implement.

There are two types of bug/fault localization approaches: static (IR-based) and dynamic (spectrum-based) approaches. *Bug localization* usually indicates the IR-based approaches, while *fault localization* usually indicates spectrum-based approaches, which involves running test cases [1]. However, some researchers use the phrase, *fault localization*, to indicate the IR-based bug localization [59]. So we have to examine the abstract and context to understand the phrase of "bug/fault localization" in the title of a paper that indicates whether IR-based or spectrum-based approaches are used.

2.1.3. Hybrid bug localization

Recent research on bug localization usually focuses on finding additional features such as version history [51,58], source code structure [2,47,50], dynamic analysis [20,30,42], and others, which could be used in ranking the relevant source-code entities. These approaches usually import one or more new features for IR-based bug localization to increase the performance. An emerging trend uses extra information in addition to textual similarity to help

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=120704.

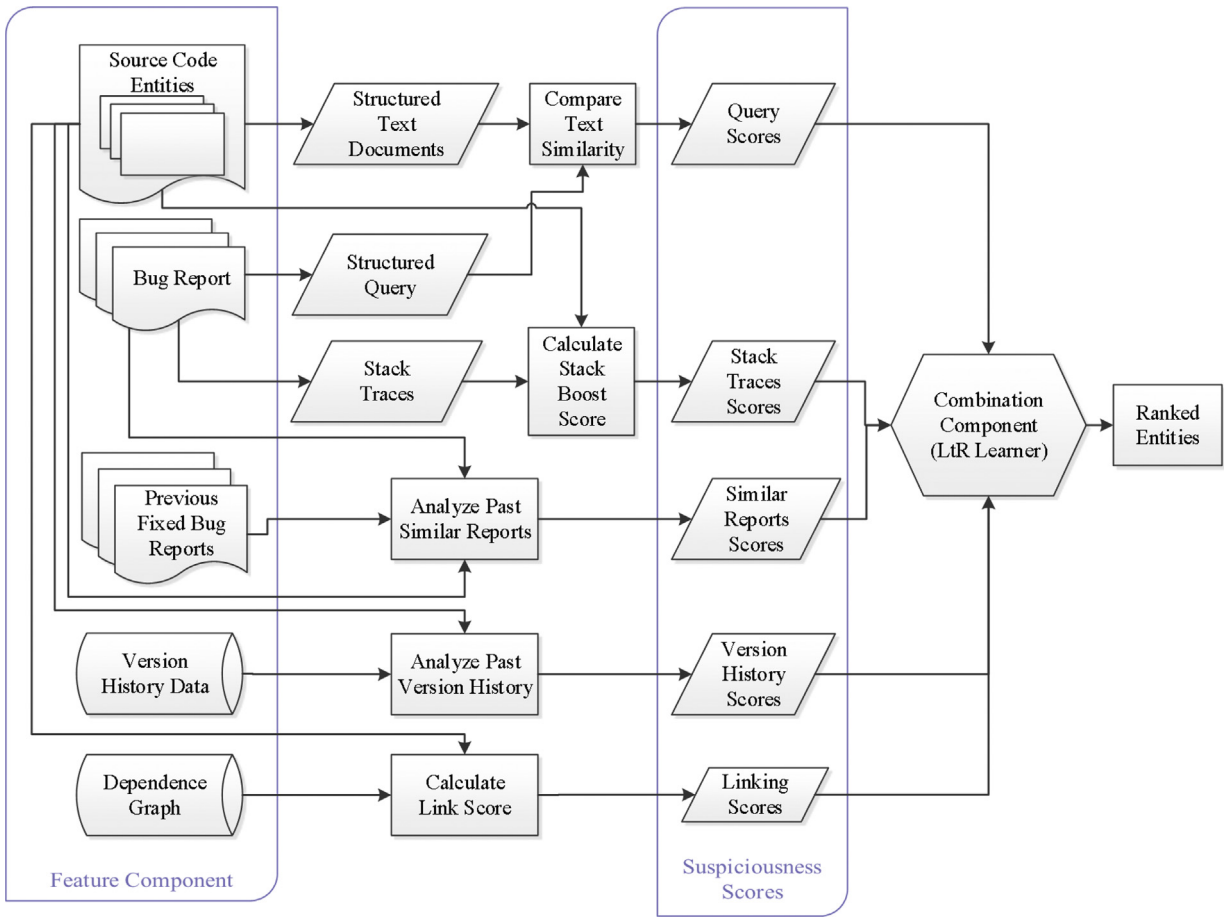

Fig. 2. Flowchart of LtR-based hybrid bug localization.

locate bugs because these features have positive effects on the efficiency of bug localization approaches [58]. Such approaches utilizing more features than textual similarity in bug localization are called hybrid bug localization. Conversely, the conventional IR-based bug localization only uses the textual similarity.

Mainly seven types of features are used by hybrid bug localization approaches. A summary of these features is shown in Table 1. For each feature in this table, a simple description is presented along with an explanation of why this feature can help locate bugs. Recent hybrid bug localization methods and the types of features that they use are shown in Table 2. These hybrid methods typically use two or three features to enhance the locating procedure, and the main feature is the textual similarity between source code entities and bug reports. The task phrases (feature location, bug localization, or fault localization) that researchers used are shown in the third column of Table 2.

In some hybrid bug localization methods [52,58,59,64,67], features are usually linearly combined with preset weight values, but the linear weights are set to fixed values from 0 to 1 to show the test performance without weight learning process. For example, two feature scores x and y are linearly combined to derive the final score. The final score of file f associated with report r is calculated as follows:

$$FinalScore(f, r) = \alpha x + (1 - \alpha) y, \quad (1)$$

where α is configured to vary from 0 to 1, with 0.1 increments. Then, the authors show the bug localization performance with a different α value. If there are three features with two linear weight α and β, α value is first optimized with $\beta = 0$ to avoid the impact of the third feature. Then the β value will be optimized with the fixed optimized α value [58,64]. However, we use LtR algorithms to combine various beneficial features. Additionally, the weight values of these features are learned from historical bug data.

The flowchart of LtR based hybrid approach is shown in Fig. 2. The left part of the flowchart contains selected feature components that include the bug report, source code entities, and other useful data for bug localization. Given a bug report, the first step is to obtain the suspiciousness score for each component. The suspiciousness scores indicate the probability of a source code entity being related to the bug. Each feature component usually outputs one suspiciousness score, except for the text information and text structure, which will be introduced in Section 2.2.1 & 2.2.5. The combination component is used to combine all the suspiciousness scores and to produce the ranked list of entities. When LtR is used as the combining technique, the combination component is a trained LtR learner. In this flowchart (Fig. 2), all features are related to the source code entities, whereas version history scores and linking scores are independent of the new bug report. In the experiment, given that the dataset provides file-level data, we set the file as the localization granularity.

2.2. Features

All the features used in the experiment are introduced in this section. They are summarized in Table 3. In short, the source code files and bug report repository, including the textual report, open date, fixed date, and related source code entities, are needed for the input part of this LtR approach. We use *component* to indicate the input data type and *attribute* to indicate the feature values (com-

Table 1
Input features in hybrid bug localization approaches.

Input data	Description	Usage for bug localization
Text information	Textual information of bug reports and source code entities.	Source code entities that are textual similar to the query tend to be related to the bug [32].
Stack trace	Some bug reports contain stack traces when the software crashes.	Entities in the stack are likely to be buggy [48,59], especially the entity in the first frame of a stack trace.
Version history	This refers to the records of the revision history of source code entities.	Revised entities in recent versions are likely to be buggy [51,58]. Bugs reoccur easily in the same location.
Dependence	This refers to the dependence graph of source code methods and classes.	Buggy entities tend to have special ranges of web scores such as PageRank and HITS [20].
Text structure	The terms in source code can be classified into different fields such as class name, method name, variable name, comment, and so on. Bug reports have title and description fields.	Different fields have various degrees of importance. For example, class names are more likely to have a common vocabulary with the bug reports than other code locations [36].
Similar report	This refers to previously fixed bug reports that are textual similar to the current bug report and their relevant source entities.	The buggy methods related to the previous similar bug reports tend to be buggy for the current report [16,58], because similar errors often have the same reason.
Execution	This refers to execution traces when testing or debugging the software.	Entities in the execution traces, which trigger the bug but do not occur in the non-bug execution traces, are likely to be related to the bug [30,42].

```
!ENTRY org.eclipse.ui 4 0 2005-09-22 14:09:47.664
!MESSAGE java.lang.NullPointerException
!STACK 0
java.lang.NullPointerException
at org.eclipse.ui.internal.PartSashContainer.drag(PartSashContainer.java :1034)
at org.eclipse.ui.internal.dnd.DragUtil.getDropTarget(DragUtil.java:338)
...
at org.eclipse.core.launcher.Main.main(Main.java:952)
```

Fig. 3. A stack trace example in the bug report #110354 in Eclipse.

ponent scores) of our learner. The execution feature is not used in our experiment since it cannot be derived in the current dataset.

2.2.1. Text information

The text information feature indicates the text similarity between source code files and bug reports. In this experiment, the text similarity between source code entities and bug reports are calculated in the same manner as BLUiR [45], which is a BM25-based algorithm. Two parameters are used in BM25 to control the scale of both term frequencies and document lengths. Given that we use the same dataset as does BLUiR, we simply use the same values of the best tuning parameters as BLUiR for simplicity. The text terms in the source code are divided into four fields: *class*, *method*, *variable*, and *comment*. Class/method/variable fields contain text terms from class/method/variable names that exist in this source code file. The comment field contains all terms from comments. To obtain all terms in the source code, we also add another source code field that is named *content*. The *content* field includes all text terms in a source code entity. This field can be considered as a weight balance because it contains all terms that belong to the other four source code fields (class, method, variable, and comment) and the other terms that do not belong to the four fields. Bug report text is divided into two fields: *summary* and *description*. After dividing the source code and bug report fields, the influence of fields such as the summary in the report and the class in the source code, which has relatively fewer terms, is enhanced to improve localization performance [45].

2.2.2. Stack trace

Sometimes a bug report contains the stack trace of the failing thread as shown in Fig. 3. A stack trace shows the sequence of instructions executed up to a crash [5]. It can also be considered as a ranked list of source code files which are potentially relevant to the bug. The intuition of using stack traces is that the file with higher rank in the stack trace gets higher boost score, which means this file is more suspicious to be buggy. We use the boost score of stack traces in the experiment as Wong et al. [59], which is the reciprocal of the file rank in the stack:

$$BoostScore(x) = \begin{cases} \frac{1}{rank} & x \in D \cap rank \leq 10 \\ 0.1 & x \in D \cap rank > 10 \\ 0.1 & x \in C \\ 0 & otherwise \end{cases}, \quad (2)$$

where x indicates the source code file, D represents the files mentioned in the stack trace, and C is the closely related files of D. We extract stack traces from bug report descriptions with Java regular expressions, which is similar to the method used by Bettenburg et al. [5]. Each entity in the stack trace is recognized as follows:

Each entity in the stack trace is recognized as follows:

```
at [METHOD] ([FILENAME].java: [LINE])
```

2.2.3. Version history

The intuition of using version history component is: if a method is just implemented or modified in recent versions, it has a high probability of causing bugs, because bugs are usually created from new functionalities or recent buggy partitions which were just modified. The version history component of our work adopts the same formula as AmaLgam [58]. The suspiciousness score is calculated with the following equation:

$$History\ Score(f, k, R) = \sum_{c \in R \wedge f \in c} \frac{1}{1 + e^{12(1-(\frac{k-t_c}{k}))}} \quad (3)$$

In the preceding equation, c indicates a single commit, f is current testing file, R refers to the set of relevant commits, k is a scale parameter that is set to 15 in the original approach, and t_c is the number of days that have elapsed between a commit c and the input bug report. The output of this algorithm is a set of suspiciousness scores, one for each file.

A certain degree of difference exists between the feature we used and AmaLgam. AmaLgam collects commits logs from the version control system of each project. For simplicity, our work only considers the bug reports and their related source code files in the BugLocator dataset as the source of version history component and treats every previously fixed bug as commits. In other words, when examining current testing file f, the old bug reports that are related to this file are considered as the relevant commits R. The fixed time of relevant bug reports is considered as the commit submitting time. Thus, we have less information on this component when compared to the original work.

Table 2
Summary of hybrid methods in bug localization.

Author and Year	Approach name	Feature/Bug/faUlt	Input data							Granularity		Combining Method		
			Text information	Stack trace	Version history	Dependence	Text structure	Similar report	Execution	Method/ Function	Class/File	Linear combination	Filter	Other
Liu'07 [30]	SITIR	F	•*	.	.	.	.	.	•*	•	.	.	•	.
Poshyvanyk'07 [42]	PROMESIR	F	•*	.	.	.	.	.	•*	•	.	•	.	.
Nichols'10 [39]	.	B	•*	.	.	.	.	•*	.	•	.	.	.	•[1]
Rao'11 [43]	.	B	•*[2]	.	.	.	.	.	.	.	•	•	.	.
Nguyen'11 [38]	BugScout	B	•*	.	.	.	.	•*	.	.	•	.	.	•[3]
Zhou'12 [67]	BugLocator	B	•*	.	.	.	.	•*	.	.	•	•	.	.
Sisman'12 [51]	.	B	•*	.	•*	.	.	.	.	.	•	.	.	•[4]
Davies'12 [16]	.	B	•*	.	.	.	.	•*	.	•	.	.	•	.
Lal'12 [27]	.	U	•*	.	.	.	•*	.	.	.	•	•	.	.
Saha'13 [45]	BLUiR	B	•*	.	.	.	•*	.	.	.	•	•	.	.
Tantithamthavorn'13 [52]	.	B	•*	.	•*	.	.	•	.	.	•	.	•	.
Dit'13 [20]	.	F	•*	.	.	•*	.	.	•	•	.	.	•	.
Moreno'14 [37]	Lobster	B	•*	•	.	•*	.	.	.	.	•	•	.	.
Wang'14 [58]	AmaLgam	B	•*	.	•*	.	•	•	.	.	•	•	.	.
Wong'14 [59]	BRTracer	U	•*	•*	.	.	•	•	.	.	•	•	.	.
Ye'14 [62]	.	B	•*	.	•	.	•	•	.	.	•	•[5]	.	.
Le'15 [28]	AML	B	•*	.	.	.	.	.	•*	•	.	•[6]	.	.
Dilshener'16 [18]	.	B	•*	•	.	.	•*	.	.	.	•	•	.	.
Wang'16 [57]	AmaLgam+	B	•*	•	•*	.	•	•	.	.	•	•[7]	.	.
Ye'16 [63]	.	B	•*	.	•	•	•	•	.	.	•	•[5]	.	.
Youm'17 [64]	BLIA	B	•*	•	•	.	•	•*	.	•	•	•	.	.

The star indicates the location of the paper summary in Section 3 according to Input data. For example, • of Liu'07 [30] exists in the Execution column of Input data. Thus, a summary of this paper is introduced in the Execution subsection of Section 3.

[1] Textual terms from old bug reports are added to related historical buggy methods.

[2] This approach combined similarity scores from various IR techniques including UM, VSM, LSA, LDA, and CBDM (Cluster Based Document Model).

[3] The entire algorithm is an integrated topic model

[4] The combining technique used the Bayesian probability, where the prior probability is from version history, the likelihood is from the IR technique, and the posterior probability is the final score.

[5] The two approaches use SVM^{Rank} [23] to learn linear weight values

[6] AML uses a probabilistic learning approach to train the linear weights

[7] AmaLgam+ uses the genetic algorithm (GA) to tune the weights of the linear combination by analyzing fixed bugs.

Table 3
The summary of attributes used in the experiment.

Input data type	Attribute			Description/Formula
Text information with structure: (2 report fields × 5 entity fields)	Bug report summary Bug report description	×	Class Method Variable Comment Content	BM25 based similarities used in BLUiR [45]
Stack trace	Stack trace score			$Boost\ Score(x) = \begin{cases} \frac{1}{rank} & x \in D \cap rank \le 10 \\ 0.1 & x \in D \cap rank > 10 \\ 0.1 & x \in C \\ 0 & \text{otherwise} \end{cases}$
Version history	Version history score			$History\ \ Score(f,k,R) = \sum_{c \in R \wedge f \in c} \frac{1}{1+e^{12(1-(\frac{k-t_c}{k}))}}$
Dependence graph	PageRank score HITS: Hub score HITS: Authority score			PageRank [9] HITS [25]
Similar report	Similar report score			$SimiScore\left(F_j\right) = \sum_{All S_i \text{ that connected to } F_j} \left(Simialrity\left(B, S_i\right)/n_i\right)$

2.2.4. Dependence

Source code methods and files have call dependencies among each other. They are similar to the hyperlinks among web pages. Since the IR based bug localization is like web search engine, the web link based ranking algorithms can also be used in bug localization. The dependence component in our approach uses PageRank [25] and HITS [9] (including the hub score and the authority score) scores as three attributes in the LtR learner. Other attributes that indicate the dependence structure of source code can also be considered.

The basis of PageRank and HITS are links between source code files. The links in our experiment are defined as follows: the link from file A to file B exists, when at least one class/method/variable defined in file B is instantiated/implemented/called in file A. The source elements with high Hub scores indicate that they are positioned in the important Hub position, which is called frequently and is connected to a large number of other modules; thus, these source elements with high Hub scores tend to be buggy easily.

2.2.5. Text structure

This component should be considered together with the IR similarity component. Given that the text information component also uses the same approach as BLUiR [45], the similarity scores between each type pair are the same as those in BLUiR. A total of 10 combinations of field pair similarities are in our algorithm, as follows: five source code fields (class, method, variable, comment, and content) and two bug reports fields (summary and description). The details of each field have been introduced in Section 2.2.1. We do not create a bug content field (including summary and description) because the number of words in the description is usually much larger than that in the summary, and the summary terms can be easily drowned in a flood of descriptive terms when the two fields are combined. The difference between this approach and BLUiR is that we use LtR techniques to combine these text similarity scores and other attributes instead of simply summing these scores. Unlike other non-textual attributes in the experiment, the essence of the text structure component is the similarity scores from different text type pairs.

2.2.6. Similar report

For a new report, its similar bugs that have been fixed and their relevant files are examined to adjust the order of the final list. Similar bug reports usually have the similar relevant buggy files [67]. Therefore, the relevant files of similar reports are ranked higher in this component. In BugLocator [67], when a new report B arrives, the textual similarities between B and all previously fixed reports S are calculated. Then, the similar report component score of the file F_j is derived as the following equation:

$$SimScore\left(F_j\right) = \sum_{All\ S_i\ connected\ to\ F_j} \frac{Similarity\left(B, S_i\right)}{n_i} \tag{4}$$

All S_i that are connected to F_j means all the bugs that were previously fixed by modifying the source code file F_j, n_i is the number of files that are modified to fix the bug S_i. Other similar report components in previous approaches are not as popularly used as the BugLocator. Thus, we opt to use BugLocator as our similar report component.

To have a unified similarity measure, the similarity measure used in this component uses the same measure in the text information component, which is based on BLUiR [45]. The difference is that we consider the bug summary and description as the two fields in the text similarity, and use the sum of each field pair similarity as the report similarity between the new report B and previous report S.

2.3. Learning to rank

In general, LtR algorithms are the methods that use machine learning technologies to solve the problem of ranking [31]. Conventional ranking models in IR include vector space model (VSM), language model (LM) [33], BM25 [44], and others; these models usually provide a similarity score between the query and the document with given parameter values. LtR techniques train learning models on these conventional IR similarities and other features such as PageRank [9] from link structure of documents, user feedback, and so on, to predict the relevance between the documents and the query in test data. Ye et al. [63] used the support vector machine (SVM) based ranking method to map bug reports to relevant files. A total of 19 features are selected to learn the SVM model. This SVM ranking method was also compared with two bug localization methods, BLUiR and BugLocator, and showed better performance. Zhao et al. [66] defined three effort-aware metrics, which are all based on lines of code (LOC), to examine the actual performance of LtR bug localization, and claimed that the LtR method is similar or even worse than the standard VSM.

LtR algorithms are grouped into three approaches: pointwise, pairwise, and listwise [31]. The three approaches model the process of LtR in different ways, and they define various input and output

Table 4
Three group approaches of LtR algorithms.

Approach	Input space	Output space	Algorithms
Pointwise	A feature vector of each single document	The relevance degree of each single document	Random Forests [8], MART [22]
Pairwise	A feature vector for each pair of documents	The pairwise preference, which takes values from {+1, −1} between each pair of documents	RankNet [11], RankBoost [21], LambdaMART [60]
Listwise	A set of documents associated with query	The ranked list (or permutation) of the documents	ListNet [12], AdaRank [61], Coordinate Ascent [34]

spaces. The details of the three types of approaches are shown in Table 4.

The **pointwise** approach uses the feature vector of a document to predict the relevance score of the document. This approach does not consider the interdependency between documents. If the number of relevant documents for each query varies considerably, the overall prediction will be dominated by queries that have a large number of relevant documents. Given that the ranking problem is more about predicting the relative order rather than the accurate relevance score, the pointwise approach is thus limited.

The **pairwise** approach uses a pair of documents to predict the relative order between them. This approach only considers the relative order between a pair of documents. The position of the documents in the final ranked list can hardly be derived when the focus is only on a pair of documents.

The **listwise** approach takes a set of documents and outputs the ranked list for each query. This approach is more in accordance with the ranking task than the pointwise and pairwise approaches. However, the training complexities of listwise approaches are usually high, and the use of the position information in some listwise ranking algorithms is insufficient [31].

3. Survey of hybrid methods

We introduce 21 hybrid bug location approaches in Table 2 and discuss 7 input feature types in detail. The hybrid methods and other related work are introduced in the subsections according to the star (*) location in Table 2.

3.1. Text information

Several information retrieval techniques have been used in report-oriented bug localization. BLUiR [45] and our previous work [49] showed that BM25 has good performance when used in bug localization. BM25 [44] is a bag-of-words ranking function implemented in Okapi information retrieval system. Each query term has a score that depends on the occurrence of this term in all documents, regardless of the interrelationship between the query terms within a document.

Rao et al. [43] compared five text models in feature location: Unigram model (UM), VSM [43], latent semantic analysis (LSA) [41], latent Dirichlet allocation (LDA) [7], cluster based document model (CBDM), and combined similarity scores from these IR techniques. The results indicate that sophisticated models such as LDA, LSA, and CBDM perform worse than the simple models such as UM and VSM. For example, in a linear combination model with LDA and UM, worse performance was observed as the weight of the LDA was increased. Thomas et al. [53] studied the effect of different configurations of VSM, LSI, and LDA on bug localization and found that the configuration is important whether in the individual model or in the proposed combination framework.

Ye et al. [62] used a LtR algorithm SVMRank [23] to find files that are relevant to bug reports. The authors extracted a new dataset by using the automatic method provided by iBugs [14]. The additional features, besides IR similarity, include API lexical similarity, bug-fixing recency, collaborative filtering score, and others. All attributes are normalized using the feature scaling method. There are more features used in the new version of this approach [63] which outperforms the old one. Totally 19 features are used including the structured text information in BLUiR [45], HITS [9] and PageRank [25] scores which have not been used in the old approach.

3.2. Stack trace

When software crashes and throws a stack trace, the first activity used to locate the bug is to check the entities in the stack trace. Schroter et al. [48] concluded that almost 60% of fixed bug reports with stack traces could be fixed by modifying the methods in the stack traces. The authors note that 40% of modified methods could be found in the first frame of stack traces, and almost 90% of modified methods could be found in the top 10 frames of stack traces.

Wong et al. [59] added source code segmentation to filter noise based on BugLocator [67]. This technique segments a file into several parts, compares the similarity between the bug report and each part, and uses the highest one as the file similarity score. The authors also used a stack trace feature to increase the performance of their technique. Thung et al. [54] proposed an integrated tool BugLocalizer based on BugLocator algorithm, which uses the previously fixed bugs.

3.3. Version history

The intuition of using version history component is as follows: if a method is merely implemented or modified in recent versions, it has a large probability of causing bugs because bugs are usually created from new functionalities or recent buggy partitions that were just modified.

Sisman et al. [51] used version history to estimate the prior probability of a file to be the target of bug localization. If a file is modified to fix a bug or has implemented a feature many times in the version history, it has a large probability of being the source of similar bugs or features. An important issue is how to weigh the score from IR techniques and historical data. The author attempted several normalization ways to obtain the best result.

Tantithamthavorn et al. [52] improved the BugLocator method by considering co-change histories. They leverage the assumption that if a buggy file is fixed, then the files changed together should be fixed together. This approach creates a co-change matrix for files, which records the number of two files that are modified together in the history. The values in the matrix are normalized using feature scaling. The file is added to the Top N list if the co-change value is higher than a threshold and the co-changed file is at the Top N list.

AmaLgam [58] put version history, similar report, and source code structure together to improve bug localization. It provided a modified function based on Google's approach [29] for version history component, which mainly depends on the period between the previous modifying time of the file and the time of input bug report. The function is introduced in Section 2.2.3 because we adopt it in our experiment. Apart from the version history component, the similar report part of AmaLgam is from BugLocator [67], while the source code structure of AmaLgam is from BLUiR [45]. AmaLgam+ [57] is an extended version of AmaLgam. In AmaLgam+, two com-

ponents (stack trace and reporter information) are added based on AmaLgam. The stack trace score of a file is simply the reciprocal of the file rank in the stack trace like Wong's approach [59]. The intuition of the reporter information component is: a bug reporter is likely to report issues from the same/similar software components.

3.4. Dependence

Dit et al. [20] applied the idea of data fusion in feature location. The approach combines information retrieval (LSI), execution (SITIR [30]), and link analysis (HITS [9] and PageRank [25]) to improve the performance of feature location. It constructed a dependence graph of source code methods such as web pages, and examined the relationship between the PageRank and HITS scores of methods and whether the methods are buggy. The researchers concluded that filtering methods with the bottom Hub scores perform best when combined with other information because low hub-score methods usually have few connections to others, so they are more akin to getter and setter methods that are not suspicious to be buggy.

Moreno et al. [37] proposed Lobster, which combines stack traces in bug reports into textual bug localization. The authors found that the nearer the distance between the currently checked source entity and the entities in the stack trace, the more suspicious the current source entity is. The distance between two source code entities is calculated based on the dependence graph [13]. The structure similarity is obtained based on the distance between the current source entity and the stack trace in the bug report. The final similarity score is the linear combination of the textual similarity and structure similarity.

3.5. Text structure

BLUiR [45] divides the source code files into four fields (class, method, variable, and comment) and bug reports into two fields (summary and description), and then calculates the textual similarity for each pair of fields from source code files and bug reports. Finally, this technique uses the sum of various attributes as the text similarity suspicious score. For our approach, we use eight attributes to indicate the eight different field-pair similarities from BLUiR, but leave the remaining learning procedure to the LtR algorithm.

Lal and Sureka [27] presented a technique to use a character n-gram IR method to enhance the similarity between bug reports and related source code entities, which have the same filename or path. In this case, the file/path names in the bug report are considered important fields, which should be enhanced to obtain improved localization performance.

Dilshener et al. [18] proposed two novel ideas: increasing the importance of file names and treating bug reports and files individually without a "one size fits all" principle. These ideas build upon researchers' ideas, namely, stack traces [37], separating bug summary from description [45] but without requesting previously fixed bug reports.

3.6. Similar report

Nichols [39] adds terms from old bug reports to the related buggy methods. Thus, the text information of methods has an additional part that represents the relevant fixed bugs. BugScout [38] is a topic model that uses the links between previous bug reports and their relevant buggy files to predict the location to be fixed for new bugs. The results show that BugScout outperforms the original LDA model as an LDA variant that incorporates historical bug reports. Davies et al. [16] train a binary decision tree classifier for every method in source code. Each transaction represents a source code method. The features are terms, and the class labels indicate whether the document is related to the method or not. The classifiers use the term similarity between the bug reports to predict whether the new report is related to the method. BugLocator [67] is a bug localization approach that uses similar report components. Several studies [52,58,59] have used BugLocator as the component of the similar reports to locate bugs more effectively. The similar report component of BugLocator is introduced in detail in Section 2.2.6 because we use it in our experiment. BLIA [64] integrated structured text information (BugLocator [67]), code change history (AmaLgam [58]), stack trace (BRTRacer [59]) and similar report (BugLocator [67]) together, and show better performance than every used approach. The granularity of BLIA is improved from file level to method level by extending bug repository data. The linear weights are set to fixed values in the range [0.1] to show the test performance.

3.7. Execution

Although some report-oriented bug localization approaches use the execution traces to improve the performance, our goal is to build an automatic framework for bug localization, thereby indicating that after the bug report is input, the system will provide the ranked entity list or the predicted buggy entities directly. Execution traces are usually derived from running past/fail test cases. This dynamic approach requires more work rather than only providing the bug report and other attributes that can be calculated or derived from the source code repository. Thus, we do not consider this attribute in this paper.

SITIR (SIngle Trace and Information Retrieval) [30] combined IR technique LSI and dynamic method that is a single-scenario execution trace. The main idea is to filter out executed methods in predefined scenarios, which depends on the report description from the ranked methods list that is derived from LSI algorithms. PROMESIR [42] is a hybrid method that combines LSI and SPR (scenario-based probabilistic ranking). SPR [3] is a dynamic method that determines the relationship between features and entities depending on the trace of execution. AML (Adaptive Multi-modal bug Localization) [28] integrates spectrum-based bug localization, TF-IDF cosine similarity, and suspicious word score. The main idea behind suspicious word component is that words that occur in the failing execution traces are more suspicious than other words; the methods with suspicious words tend to be the buggy methods. Dao et al. [15] studied the influence of execution information (i.e. coverage, slicing, and spectrum) to IR based bug localization. They concluded that both the coverage and slicing information can improve IR-based techniques at both class and method levels, but spectrum information can only improve localization at the method level not the class level.

3.8. Discussion

Among those hybrid bug localization approaches introduced above, there is a trend that the performance is increased by combining suitable beneficial features. For example, the authors of AmaLgam [58] added a version history component to the approach based on BLUiR [45] and BugLocator [67], then AmaLgam outperforms the two based approaches. Two more components (stack trace and reporter information) were added in AmaLgam+ [57] to improve AmaLgam. Ye et al. [63] added structured textual similarity in BLUiR and web link scores to the old version approach [62] to improve the localization performance. BLIA [64] integrated four components from three other approaches and outperforms the three approaches. Thus, finding and combining more suitable beneficial features together is a manner to improve bug localization performance. However, an approach combining more features does

not always perform better [15]. Thus, choosing the suitable features and suitable combining methods is most important.

Apart from the seven main feature types discussed above, there are some other feature types used in bug localization such as reporter information component in AmaLgam+ [57], the API documents of source code [62,63], and so on. These features are not so widely used as the other features in the seven main types. Apart from the seven feature types concluded in this paper, there will be more beneficial features found and the taxonomy of feature types will also be extended in the future.

4. Experimental evaluation

4.1. Dataset

Several hybrid approaches [45,52,58,59,67] have used the BugLocator [67] dataset.[2] Thu, we can conveniently compare the performance of our approach to other hybrid approaches. Few comparisons have been made between other hybrid bug localization approaches, because those approaches usually use self-extracted datasets. The BugLocator dataset is at the file level. The details are shown in Table 5. Although the periods of the three projects are long, and many versions are found across these periods, we can only select the earliest version of each project for simplicity as the BugLocator did originally.

4.2. Tools

We use the RankLib[3] tool, which implements eight LtR algorithms, including MART (Multiple Additive Regression Trees, a.k.a. Gradient boosted regression tree) [22], RankNet [11], RankBoost [21], AdaRank [61], Coordinate Ascent [34], LambdaMART [60], ListNet [12], and Random Forests [8]. We tested all these algorithms in the experiment.

Although most of the features used in this study have been proposed, we need to develop our own tool to derive the attributes. The attributes are intermediate results that cannot be derived from the tools or results provided by the authors of hybrid methods.

JRipples [10] is an Eclipse plug-in that supports feature location. We modify the JRipples tool to extract data, including term frequency of different fields in source code and dependence graph, through JDT (Eclipse Java development tools).[4] We also use the R-project[5] to preprocess and produce each feature of the approach, including PageRank and HITS scores.

4.3. Evaluation

Three main evaluation metrics used in bug localization are TopN, MAP, and MRR:

TopN refers to the number of bugs with their related source code entities ranked in the top N (N = 1, 5, 10) of the result list [67]. For a bug report, if one of the entities that needs be modified to fix the bug appears in the top N of the ranked list, the bug report is considered as localized, and it is counted in the TopN result. The higher the TopN value, the better is the performance of bug localization methods.

Mean Average Precision (MAP) provides a single-figure measure of information retrieval quality [33] when a query has multiple related documents. Average precision (AvgP) is the average of the precision value for a single-query $q_j \in Q$, which is defined as follows:

$$AvgP_j = \frac{1}{m_j}\sum_{k=1}^{m_j} Precsion(R_{jk}) \tag{5}$$

where the set of relevant documents for query q_j is defined as $\{d_1,\ldots,d_{mj}\}$, m_j is the number of relevant documents of query q_j, R_{jk} is the set of ranked retrieval results from the top result until document d_k is obtained. The precision of R_{jk} is defined as follows, where the symbol # indicates "the number of":

$$Precision\left(R_{jk}\right) = \frac{\#\left(related\ documents\ in\ R_{jk}\right)}{\#\left(documents\ in\ R_{jk}\right)}, \tag{6}$$

MAP is the mean of AvgP in all queries as follows:

$$MAP = \frac{1}{|Q|}\sum_{j=1}^{|Q|} AvgP_j\ , \tag{7}$$

where |Q| is the number of queries. A higher MAP value indicates better performance.

Mean reciprocal rank (MRR) is a statistical measure that depends on the first related document in the ranked list for a query [55]. The reciprocal rank of a query in bug localization is the multiplicative inverse of the effective measure, where effective measure (EM) is the highest rank of relevant source code entities of the bug in the final ranked list [42]. MRR is the mean of reciprocal rank for all queries and is expressed as follows:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{EM_i} \tag{8}$$

The higher the MRR value is, the better the method performs.

4.4. Comparison to reproduced work

The LtR approach uses the same features of related studies; thus, we can reproduce the BLUiR [45] and AmaLgam [58] results based on our extracted features. AmaLgam uses a linear combination of text similarity, similar report, and version history as the final ranking score of a source code entity. The parameters of our reproduced AmaLgam have been tuned to obtain the best performance as introduced in the original paper [58]. The BLUiR approach uses the sum of eight features as the suspicious score of a source code file being relevant to a bug report, and the eight features are eight combinations of BM25-based similarities (two bug report fields combined with four source-code fields). The AmaLgam approach uses the same eight text similarity features as BLUiR, with an additional two features including the version history and stack trace. Our LtR approaches use the same 10 features as AmaLgam with additional 6 features as shown in Table 6.

Although we use some features that have been proposed in BLUiR and AmaLgam, AmaLgam does not provide results or tools. BLUiR only provides the final result without including the features we needed in our LtR techniques. Thus, we developed our own tool to produce these features. However, some differences are observed between our development and the original authors. For example, in text preprocessing and extracting, a slight modification in some steps will cause differences. The performance of our reproduced BLUiR and AmaLgam are different from those claimed by the original authors. Thus, we did not use the original evaluation result of BLUiR and AmaLgam. However, because our LtR approach is based on the same features of reproduced techniques, comparing the performance of our LtR approach and the reproduced techniques is fair.

[2] The dataset website is at https://code.google.com/p/bug/.
[3] http://sourceforge.net/p/lemur/wiki/RankLib/.
[4] http://www.eclipse.org/jdt/.
[5] http://www.r-project.org/.

Table 5
Details of BugLocator dataset.

Project	Description	Period	#Bugs	#Files
SWT-3.1	Widget toolkit for Java	Oct 2004–Apr 2010	98	484
ZXing	Barcode image processing for Android	Mar 2010–Sep 2010	20	391
Eclipse-3.1	Popular IDE for java	Oct 2004–Mar 2011	3075	11854

Table 6
Features used in BLUiR, AmaLgam, and LtR approach.

Feature(s)		BLUiR	AmaLgam	LtR
8-field combination of text similarity: source code (class, method, variable, and comment) * bug report (summary and description)		Yes	Yes	Yes
Similar report		No	Yes	Yes
Version history		No	Yes	Yes
2-field combination of text similarity: source code (content) * bug report (summary and description)		No	No	Yes
Stack trace		No	No	Yes
Dependence graph	PageRank	No	No	Yes
	HITS: Hub	No	No	Yes
	HITS: Authority	No	No	Yes

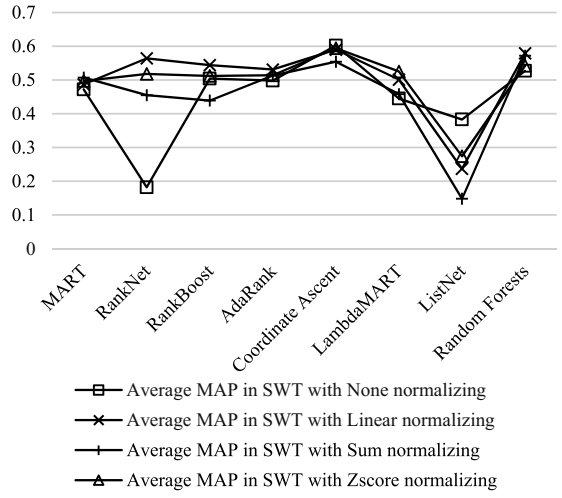

Fig. 4. Average MAP of SWT project on the test data in 5-fold cross validation.

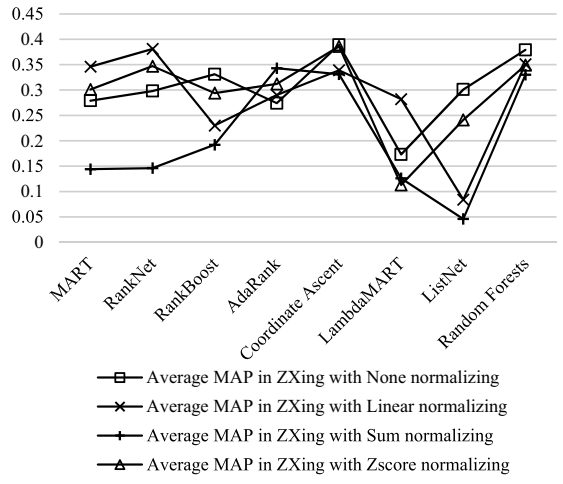

Fig. 5. Average MAP of ZXing project on the test data in 5-fold cross validation.

4.5. SWT and ZXing result

In the RankLib parameter setting, we use 5-fold cross validation, and test on all normalizing setting including none normalizing, linear (normalizing each feature by its min/max values), sum (normalizing each feature by the sum of all its values), and Zscore (normalizing each feature by its mean/standard deviation). In the five chunks of data, three are considered as the train data, one chunk is for validation to avoid overfitting in the train set, and the last chunk is the test data. Recent bug localization approaches typically use MAP, MRR, and TopN to evaluate the results. We select the MAP as the evaluation metric on test data and the metric to optimize on the training data, given that the RankLib evaluation metric does not support MRR and TopN. For the rest of the parameters, we use the default setting for each algorithm.

Tables 7 and 8 show the average MAP result of two projects in 5-fold cross validation. The coordinate ascent algorithm [34] with none normalizing method has the highest MAP values in the test set for both projects, which are highlighted by the bold type in each table. The train/test column indicates the performance value on the train/test data. The average MAP values on the test data in two tables are also shown in Figs. 4 and 5. The coordinate ascent algorithm and random forest have good performances in both projects and with all four normalizing methods. Compared with the high MAP values on the train data, the random forests algorithm has relatively low performance on the test data because of overfitting. ListNet has worst performance on average.

The normalizing methods have different influences on the test MAP values in two projects. In SWT project, the MAP values of four normalizing methods are closed to each other when using the same LtR algorithm, except for none normalizing using RankNet. In ZXing project, the MAP values of four normalizing methods have a larger difference than the ones in SWT project, since ZXing project only contains 20 bugs and only 4 bugs are used to test in each fold. AdaRank, coordinate ascent, and random forests have similar MAP values with different normalizing methods respectively, that is, there is little influence of selecting different normalizing methods in the three rankers.

4.6. Eclipse result

We considered the Eclipse project, which is large with 3075 bugs and 11,854 source code files. To save running time, we divide the bugs into 30 folds in chronological order and use a sliding window setting: one fold as the train set and the next fold as the test set. We use the coordinate ascent algorithm without normalization because it has shown good performance in SWT and ZXing projects in general. To get better performance, a parameter tuning process is added in Eclipse testing, although default parameter values are

Table 7
Average MAP result of SWT project in 5-fold cross validation. (The highest MAP value on the test data is highlighted).

Normalization	None		Linear		Sum		Zscore	
Ranker \Train or Test	Train	Test	Train	Test	Train	Test	Train	Test
1 MART	0.599	0.472	0.644	0.487	0.681	0.507	0.668	0.497
2 RankNet	0.180	0.182	0.552	0.564	0.422	0.455	0.499	0.518
3 RankBoost	0.509	0.504	0.547	0.544	0.500	0.439	0.528	0.512
4 AdaRank	0.520	0.498	0.558	0.531	0.535	0.513	0.545	0.514
5 Coordinate Ascent	0.621	**0.602**	0.634	0.590	0.614	0.554	0.633	0.594
6 LambdaMART	0.580	0.445	0.635	0.501	0.677	0.458	0.648	0.525
7 ListNet	0.392	0.383	0.224	0.237	0.129	0.148	0.296	0.274
8 Random Forests	0.981	0.527	0.982	0.579	0.980	0.572	0.983	0.542

Table 8
Average MAP result of ZXing project in 5-fold cross validation. (The highest MAP value on the test data is highlighted).

Normalization	None		Linear		Sum		Zscore	
Ranker \Train or Test	Train	Test	Train	Test	Train	Test	Train	Test
1 MART	0.698	0.279	0.758	0.346	0.696	0.144	0.744	0.301
2 RankNet	0.253	0.298	0.389	0.381	0.182	0.146	0.371	0.347
3 RankBoost	0.477	0.331	0.379	0.230	0.319	0.192	0.413	0.294
4 AdaRank	0.348	0.274	0.370	0.290	0.377	0.343	0.366	0.312
5 Coordinate Ascent	0.521	**0.389**	0.506	0.339	0.517	0.331	0.535	0.386
6 LambdaMART	0.769	0.173	0.780	0.282	0.601	0.126	0.715	0.113
7 ListNet	0.200	0.301	0.109	0.084	0.168	0.046	0.245	0.241
8 Random Forests	0.977	0.379	0.975	0.351	0.981	0.330	0.978	0.349

Table 9
Performance comparison of LtR-based approach and reproduced state-of-art techniques. (The highest value of each eavaluation metric is highlighted).

Project	Evaluation metric	LtR (coordinate ascent without normalization)	BLUiR	AmaLgam
ZXing	Top1	6	7	**8**
	Top5	14	15	**16**
	Top10	15	**17**	**17**
	MAP	0.389	0.412	**0.447**
	MRR	0.459	0.499	**0.532**
SWT	Top1	**55**	44	46
	Top5	**93**	92	91
	Top10	**120**	117	120
	MAP	**0.602**	0.505	0.526
	MRR	**0.671**	0.576	0.598
Eclipse	Top1	**914**	828	870
	Top5	**2043**	1934	1975
	Top10	2615	2554	**2616**
	MAP	**0.306**	0.278	0.289
	MRR	**0.399**	0.374	0.389

used in SWT and ZXing projects. First, two parameters of coordinate ascent algorithm are tuned using grid search: restarts = (3, 5, 10, 20) and iterations = (10, 25, 50, 100). The grid search values are selected reasonably with the default parameter value in the middle. We trained the algorithm on the first fold, validated it on the second fold, and tested it on the third fold. The best model on the validation set is selected to test. We found that the algorithm performed best on the third fold of Eclipse using the default parameter (restarts = 5, iterations = 25). Thus, the default parameters are used in the following evaluation steps:

For the Eclipse project, there are numerous irrelevant files for each bug. Therefore, we first used the BLUiR method to rank all files for each bug, and selected the top 200 irrelevant files and all relevant files for each bug as the train set. The results of applying the coordinate ascent algorithm on the Eclipse project is shown in Fig. 6. The train curve indicates the MAP values evaluated on the train fold itself. The test curve represents the MAP values tested on the next fold of the training fold. We also attempted to add 1 fold as a validation set or split the train set for validation. However, one or two folds perform poorly when the validation set is added, and their average MAP values are closed to the ones in the current presented setting (one fold for train and next fold for test). The BLUiR

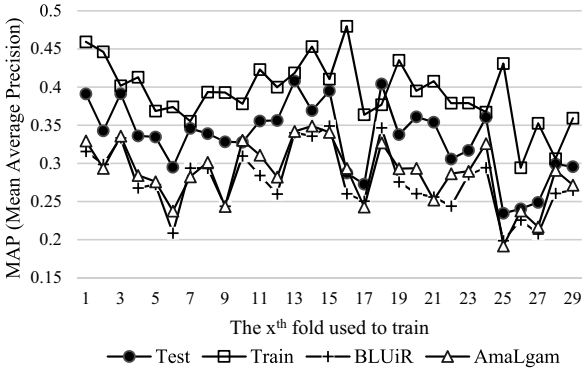

Fig. 6. MAP of using Coordinate Accent in 30 folds of Eclipse project.

[45] and AmaLgam [58] curves show the performance of reproduced corresponding algorithms in each fold. The BLUiR method does not need to be trained further, and the parameters of AmaLgam is tuned by maximizing the average MAP values of all bugs. We applied Wilcoxon signed-rank test on the MAP values of coor-

dinate ascent, BLUiR, and AmaLgam in each fold. The result shows that coordinate ascent is significantly better than BLUiR (p = 2.5e−6) and AmaLgam (p = 3.9e−6).

4.7. Result comparison

Given that the coordinate ascent algorithm with no-normalizing performs best in our previous experiments as shown in Tables 7 and 8, we select it for comparison with reproduced BLUiR and AmaLgam approaches in the MAP, MRR, and TopN (N = 1, 5, 10) metrics, which is shown in Table 9. The evaluation results of LtR are derived from the entire project that combined each fold-test set. Unlike SWT and ZXing projects in which the test data are trained and validated on the other four folds, the Eclipse test data are trained on the previous fold in the bug-fixed time order, and the first fold is trained on the last fold. The results of BLUiR and AmaLgam are directly from the entire project because no training parts are present. The results show that the LtR outperforms BLUiR and AmaLgam on the large projects, Eclipse and SWT, by using coordinate ascent without normalization. The performance of LtR is no better than the other two techniques that consider the ZXing project. However, ZXing has only 20 bugs, with 16 bugs used for training and the remaining 4 bugs used for testing in the 5-fold LtR approaches. The small data size of ZXing could be the main reason why LtR did not exhibit good performance. We found that the linear weights of ZXing differed much more in each fold than did the other projects, thereby indicating that the results of ZXing easily overfit the train set.

By contrast, the data of the Eclipse project have 3075 bugs and many source code files. Even when the project is divided into 30 folds, more than 100 bugs are found for training in each fold. Fig. 6 indicates that the LtR test performances in most folds are better than or similar to the other two reproduced techniques.

5. Threats to validity

Our approach faces the following threats to validity:

First, we only use the default parameter setting for each LtR algorithm in RankLib. The performance of LtR algorithms should be different when different parameter values are used. The next step of this study is testing the influence of various parameter settings in LtR-based bug localization.

Second, the bug repository is extracted from the version history, and the relevant files are labeled via the modification commits with text "bug|fix". Thus, the labeled relevant files may not be the real files that need to be modified to fix the bug. This threat to validity exists in all bug localization approaches that use the automatic extraction method without manual examination [26].

Third, we applied learning to rank algorithms to our selected beneficial features. Different conclusions may be derived by using different features. However, instinctively, combining more correlated features in a suitable manner could increase the localization performance. This area is left for future work.

6. Conclusion

Information retrieval-based bug localization has been studied for several years. Recently proposed hybrid bug localization methods have examined and adopted many helpful features from version history, similar bug reports, source code structure, and so on, in addition to the textual similarity. We conducted a survey of these hybrid methods and features and compared eight different LtR techniques with four normalization methods in bug localization and two reproduced state-of-art approaches. The results showed that the coordinate ascent algorithm without normalization is a suitable LtR method for our selected features, and it outperforms two state-of-art approaches for two large projects, namely, Eclipse and SWT.

Acknowledgements

This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (No. 11208017 and 11214116), the research funds of City University of Hong Kong (No. 7004683), and National Key R&D Program of China (No. 2016YFB1000303).

References

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, A.J.C. van Gemund, A practical evaluation of spectrum-based fault localization, J. Syst. Software 82 (11) (2009) 1780–1792.
[2] N. Ali, A. Sabane, Y. Gueheneuc, G. Antoniol, Improving bug location using binary class relationships. source code analysis and manipulation (SCAM), 2012, IEEE 12th International Working Conference on (2012) (2012) 174–183.
[3] G. Antoniol, Y.-G. Gueheneuc, Feature identification: an epidemiological metaphor, Software Engineering, IEEE Transactions 32 (9) (2006) 627–641.
[4] R. Beltraín-Alfonso, A. Torres-Tautiva, P.A. Gaona-Garcia, C.E. Montenegro-Marin, Exploring the relevance of search engines: an overview of google as a case study, Int. J. Interact. Multimedia Artif. Intell. 4 (4) (2017) 72–79.
[5] N. Bettenburg, R. Premraj, T. Zimmermann, S. Kim, Extracting structural information from bug reports, in: Proceedings of the 2008 International Working Conference on Mining Software Repositories, Leipzig, Germany, 2008, pp. 27–30.
[6] D. Binkley, D. Lawrie, Learning to rank improves IR in SE. software maintenance and evolution (ICSME), IEEE International Conference on (2014) (2014) 441–445.
[7] D.M. Blei, A.Y. Ng, M.I. Jordan, Latent dirichlet allocation, J. Mach. Learn. Res. 3 (4–5) (2003) 993–1022.
[8] L. Breiman, Random forests, Machine Learning 45 (1) (2001) 5–32.
[9] S. Brin, L. Page, The anatomy of a large-scale hypertextual Web search engine, Comput. Networks {ISDN} Syst. 30 (1–7) (1998) 107–117.
[10] J. Buckner, J. Buchta, M. Petrenko, V. Rajlich, JRipples: a tool for program comprehension during incremental change, Washington, DC, USA, in: Proceedings of the 13th International Workshop on Program Comprehension, 005, 2005, pp. 149–152.
[11] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, G. Hullender, Learning to rank using gradient descent, in: Proceedings of the 22Nd International Conference on Machine Learning, Bonn, Germany, 2005, pp. 89–96.
[12] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, H. Li, Learning to rank: from pairwise approach to listwise approach, in: Proceedings of the 24th International Conference on Machine Learning, Corvalis, Oregon, USA, 2007, pp. 129–136.
[13] K. Chen, V. Rajlich, Case study of feature location using dependence graph. Program Comprehension, 2000, in: Proceedings. IWPC 2000, 8th International Workshop on (2000), 2000, pp. 241–247.
[14] V. Dallmeier, T. Zimmermann, Extraction of bug localization benchmarks from history, in: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, Atlanta, Georgia, USA, 2007, pp. 433–436.
[15] T. Dao, L. Zhang, N. Meng, How does execution information help with information-retrieval based bug localization? in: Proceedings of the 25th International Conference on Program Comprehension, Buenos Aires, Argentina, 2017, pp. 241–250.
[16] S. Davies, M. Roper, M. Wood, Using bug report similarity to enhance bug localisation, in: Reverse Engineering (WCRE), 2012, 19th Working Conference on (2012), 2012, pp. 125–134.
[17] S. Deerwester, S. Deerwester, 1990. Indexing by latent semantic analysis J. Am. Soc. Inf. Sci. 41 (6) (1990) 391–407.
[18] T. Dilshener, M. Wermelinger, Y. Yu, Locating bugs without looking back, in: Proceedings of the 13th International Conference on Mining Software Repositories, Austin, Texas, 2016, pp. 286–290.
[19] B. Dit, M. Revelle, M. Gethers, D. Poshyvanyk, 2013. Feature location in source code: a taxonomy and survey, J. Software Evol. Process 25 (1) (2013) 53–95.
[20] B. Dit, M. Revelle, D. Poshyvanyk, 2013. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software, Empirical Software Eng. 18 (2) (2013) 277–309.
[21] Y. Freund, R. Iyer, R.E. Schapire, Y. Singer, An efficient boosting algorithm for combining preferences, J. Mach. Learn. Res. 4 (2003) 933–969.
[22] J.H. Friedman, Greedy function approximation: a gradient boosting machine, Ann. Stat. 29 (5) (2001) 1189–1232.
[23] T. Joachims, Training linear SVMs in linear time, in: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, 2006, pp. 217–226.
[24] J.A. Jones, M.J. Harrold, Empirical evaluation of the tarantula automatic fault-localization technique, in: Proceedings of the 20th IEEE/ACM

International Conference on Automated Software Engineering, Long Beach, CA, USA, 2005, pp. 273–282.
[25] J.M. Kleinberg, Authoritative sources in a hyperlinked environment, J. ACM 46 (5) (1999) 604–632.
[26] P.S. Kochhar, Y. Tian, D. Lo, Potential biases in bug localization: do they matter? in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, Vasteras, Sweden, 2014, pp. 803–814.
[27] S. Lal, A. Sureka, A static technique for fault localization using character N-gram based information retrieval model, in: Proceedings of the 5th India Software Engineering Conference, Kanpur, India, 2012, pp. 109–118.
[28] T.-D.B. Le, R.J. Oentaryo, D. Lo, Information retrieval and spectrum based bug localization: better together, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, 2015, pp. 579–590.
[29] C. Lewis, R. Ou, Bug Prediction at Google, 2011 http://google-engtools.blogspot.sg/2011/12/bug-prediction-at-google.html.
[30] D. Liu, A. Marcus, D. Poshyvanyk, V. Rajlich, Feature location via information retrieval based filtering of a single scenario execution trace, in: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, Atlanta, Georgia, USA, 2007, pp. 234–243.
[31] T.-Y. Liu, Learning to Rank for Information Retrieval, Springer, Berlin Heidelberg, 2011.
[32] S.K. Lukins, N.A. Kraft, L.H. Etzkorn, 2010. Bug localization using latent Dirichlet allocation Inf. Software Technol. 52 (9) (2010) 972–990.
[33] C.D. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, Cambridge University Press, Cambridge, 2008.
[34] D. Metzler, W. Bruce Croft, Linear feature-based models for information retrieval, Information Retrieval 10 (3) (2007) 257–274.
[35] A. Moreno, T. Redondo, Text analytics: the convergence of big data and artificial intelligence, Int. J. Int. Multimedia Artif. Intell. 3 (6) (2016) 57–64.
[36] L. Moreno, W. Bandara, S. Haiduc, A. Marcus, On the relationship between the vocabulary of bug reports and source code. software maintenance (ICSM), 29th IEEE International Conference on (2013) (2013) 452–455.
[37] L. Moreno, J.J. Treadway, A. Marcus, W. Shen, On the use of stack traces to improve text retrieval-Based bug localization, IEEE International Conference on Software Maintenance and Evolution (2014) 151–160.
[38] A.T. Nguyen, T.T. Nguyen, J. Al-Kofahi, H.V. Nguyen, T.N. Nguyen 2011, A topic-based approach for narrowing the search space of buggy files from a bug report, in: Automated Software Engineering (ASE), 26th IEEE/ACM International Conference on (2011), 2011, pp. 263–272.
[39] B.D. Nichols, Augmented bug localization using past bug information, in: Proceedings of the 48th Annual Southeast Regional Conference, Oxford, Mississippi, 2010, 61:1–61:6.
[40] M.F. Porter, An algorithm for suffix stripping, Prog. Electron. Lib. Inf. Syst. 14 (3) (1980) 130–137.
[41] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, V. Rajlich, Combining probabilistic ranking and latent semantic indexing for feature identification, in: Program Comprehension, 2006. ICPC 2006, IEEE International Conference on (2006), 2006, pp. 137–148.
[42] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, V. Rajlich, Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. software engineering, IEEE Transactions 33 (6) (2007) 420–432.
[43] S. Rao, A. Kak, 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, Waikiki, Honolulu,HI, USA, 2011, pp. 43–52.
[44] S.E. Robertson, S. Walker, S. Jones, M.M. Hancock-Beaulieu, M. Gatford, Okapi at TREC-3. Overview of the Third Text REtrieval Conference (TREC), 1995, pp. 1995.
[45] R.K. Saha, M. Lease, S. Khurshid, D.E. Perry, Improving bug localization using structured information retrieval. Automated Software Engineering (ASE), IEEE/ACM 28th International Conference on (Nov. 2013) (2013) 345–355.
[46] G. Salton, A. Wong, C.S. Yang, 1975. a vector space model for automatic indexing, Commun. ACM 18 (11) (1975) 613–620.
[47] G. Scanniello, A. Marcus, Clustering support for static concept location in source code. program comprehension (ICPC), IEEE 19th International Conference on (2011) (2011) 1–10.
[48] A. Schroter, N. Bettenburg, R. Premraj, Do stack traces help developers fix bugs? Mining Software Repositories (MSR), 7th IEEE Working Conference on (May. 2010) (2010) 118–121.
[49] Z. Shi, J. Keung, Q. Song, An empirical study of BM25 and BM25F based feature location techniques, in: Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices, Hong Kong, China, 2014, pp. 106–114.
[50] P. Singh, S. Batra, A novel technique for call graph reduction for bug localization, Int. J. Comput. Appl. 47 (15) (2012) 1–5.
[51] B. Sisman, A.C. Kak, Incorporating version histories in information retrieval based bug localization. mining software repositories (MSR), 9th IEEE Working Conference on (Jun. 2012) (2012) 50–59.
[52] C. Tantithamthavorn, A. Ihara, K.-I. Matsumoto, Using Co-change histories to improve bug localization performance. software engineering, artificial intelligence, networking and Parallel/Distributed computing (SNPD), 14th ACIS International Conference on (Jul. 2013) (2013) 543–548.
[53] S.W. Thomas, M. Nagappan, D. Blostein, A.E. Hassan, The impact of classifier configuration and classifier combination on bug localization. software engineering, IEEE Transactions 39 (10) (2013) 1427–1443.
[54] F. Thung, T.-D.B. Le, P.S. Kochhar, D. Lo, BugLocalizer: integrated tool support for bug localization, in: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 2014, pp. 767–770.
[55] E.M. Voorhees, in: E. Voorhees, D.E. Harman (Eds.), The TREC-8 Question Answering Track Report. Natural Language Engineering, NIST, 1999, pp. 77–82.
[56] S. Wang, F. Khomh, Y. Zou, Improving bug localization using correlations in crash reports. Mining Software Repositories (MSR), 10th IEEE Working Conference on (May. 2013) (2013) 247–256.
[57] S. Wang, D. Lo, 2016. AmaLgam+: composing rich information sources for accurate bug localization, J. Software Evol. Process 28 (10) (2016) 921–942.
[58] S. Wang, D. Lo, Version history, similar report, and structure: putting them together for improved bug localization, in: Proceedings of the 22Nd International Conference on Program Comprehension, Hyderabad, India, 2014, pp. 53–63.
[59] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, H. Mei, Boosting bug-Report-Oriented fault localization with segmentation and stack-Trace analysis, in: Software Maintenance and Evolution (ICSME), IEEE International Conference on (2014), 2014, pp. 181–190.
[60] Q. Wu, C.C. Burges, K. Svore, J. Gao, Adapting boosting for information retrieval measures, Inf. Retrieval 13 (3) (2010) 254–270.
[61] J. Xu, H. Li, AdaRank: a boosting algorithm for information retrieval, in: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Amsterdam, The Netherlands, 2007, pp. 391–398.
[62] X. Ye, R. Bunescu, C. Liu, 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, Hong Kong, China, 2014, pp. 689–699.
[63] X. Ye, R. Bunescu, C. Liu, Mapping bug reports to relevant files: a ranking model, a fine-Grained benchmark, and feature evaluation, IEEE Trans. Software Eng. 42 (4) (2016) 379–402 (Apr. 2016).
[64] K.C. Youm, J. Ahn, E. Lee, Improved bug localization based on code change histories and bug reports, Inf. Software Technol. 82 (2017) 177–192.
[65] C. Zhai, J. Lafferty, A study of smoothing methods for language models applied to information retrieval, ACM Trans. Inf. Syst. 22 (2) (2004) 179–214.
[66] F. Zhao, Y. Tang, Y. Yang, H. Lu, Y. Zhou, B. Xu, Is learning-to-Rank cost-Effective in recommending relevant files for bug localization? IEEE International Conference on Software Quality, Reliability and Security (2015) 298–303 (Aug. 2015).
[67] J. Zhou, H. Zhang, D. Lo, Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. Software Engineering (ICSE), 34th International Conference on (2012) (2012) 14–24.