Autom Softw Eng (2018) 25:383–434
https://doi.org/10.1007/s10515-017-0226-1

Locating bugs without looking back

Tezcan Dilshener[1] · Michel Wermelinger[1] · Yijun Yu[1]

Received: 29 July 2016 / Accepted: 30 August 2017 / Published online: 10 October 2017

Abstract Bug localisation is a core program comprehension task in software maintenance: given the observation of a bug, e.g. via a bug report, where is it located in the source code? Information retrieval (IR) approaches see the bug report as the query, and the source code files as the documents to be retrieved, ranked by relevance. Such approaches have the advantage of not requiring expensive static or dynamic analysis of the code. However, current state-of-the-art IR approaches rely on project history, in particular previously fixed bugs or previous versions of the source code. We present a novel approach that directly scores each current file against the given report, thus not requiring past code and reports. The scoring method is based on heuristics identified through manual inspection of a small sample of bug reports. We compare our approach to eight others, using their own five metrics on their own six open source projects. Out of 30 performance indicators, we improve 27 and equal 2. Over the projects analysed, on average we find one or more affected files in the top 10 ranked files for 76% of the bug reports. These results show the applicability of our approach to software projects without history.

This is a substantially extended version of our previous 4-page paper (Dilshener et al. 2016). We have added 2 research questions, analysed 2 more projects (Pillar1 and Pillar2), and compared against 3 more approaches (Wong et al. 2014; Youm et al. 2015; Rahman et al. 2015). Sections 4 and 5 are new (except 4.1.3 and 4.1.4) and existing sections have been considerably expanded with new material.

✉ Tezcan Dilshener
tezcan@dilshener.de

Michel Wermelinger
michel.wermelinger@open.ac.uk

Yijun Yu
yijun.yu@open.ac.uk

1 School of Computing and Communications, The Open University, Milton Keynes MK7 6AA, UK

Keywords Bug localisation · Information retrieval · Empirical study

1 Introduction

Current software applications are valuable strategic assets to companies: they play a central role for the business and require continued maintenance. In recent years, research has focused on utilising concept location techniques in bug localisation to address the challenges of performing maintenance tasks on such applications.

In general, software applications consist of multiple components. When certain components of the application do not perform according to their predefined functionality, they are classified to be in error. These unexpected and unintended erroneous behaviours, also referred as bugs, are known to be often the product of coding mistakes. Upon discovering such abnormal behaviour of the software, a developer or a user reports it in a document referred as bug report (BR). BR documents may provide information that could help in fixing the bug by changing the relevant program elements of the application. Identifying where to make changes in response to a BR is called bug localisation. The change request is expressed as a BR and the end goal is to change the existing program elements (e.g. source code files) to correct an undesired behaviour of the software.

Li et al. (2006) classified bugs according to three categories: root cause, impact and the involved software components.

– The root cause category includes memory and semantic related bugs like improper handling of memory objects or semantic bugs, which are inconsistent with the original design requirements or the programmers' intention.
– The impact category includes performance and functionality related bugs, like the program keeps running but does not respond, halts abnormally, mistakenly changes user data, or functions correctly but runs/responds slowly.
– The software components category includes bugs related to the components implementing core functionality, graphical user interfaces, runtime environment and communication, as well as data base handling.

The study in Li et al. (2006), performed with two open source software (OSS) projects, showed that 81% of all bugs in Mozilla and 87% of those in Apache are semantics related. These percentages increase as the applications mature, and they have direct impact on system availability, contributing to 43–44% of crashes. Since it takes a longer time to locate and fix semantic bugs, more effort needs to be put into helping developers locate the bugs.

Generally, program comprehension tasks during software maintenance require additional effort from those developers who have little domain knowledge (Bennett and Rajlich 2000; Starke et al. 2009; Antoniol et al. 2002; Abebe et al. 2011). Therefore, a programmer can easily introduce semantic bugs due to inconsistent understanding of the requirements or intentions of the original developers. Early attempts to aid developers in recovering traceability links between source code files and system documentation used Information Retrieval (IR) methods, such as the Vector Space Model (VSM) (Salton and Buckley 1988) and Latent Semantic Indexing (LSI), and managed to achieve high precision (Marcus and Maletic 2003) or high recall (Antoniol et al.

2002). The idea behind VSM is that the more times a query term appears in a document relative to the number of times the term appears in all the documents in the collection, the more relevant that document is to the query. Vectors represent queries (bug reports in the case of bug localisation) and documents (source code files). Each element of the vector corresponds to a word or term extracted from the query's or document's vocabulary. The relevance of a document to a query can be directly evaluated by calculating the similarity of their word vectors.

These probabilistic IR approaches do not consider terms that are strongly related via structural information and thus still perform poorly in some cases (Petrenko and Rajlich 2013). Pure textual similarity may not be able to distinguish the actual buggy file from other files that are similar but unrelated (Wang et al. 2015). For example, Moreno et al. (2014) noted that when two files are structurally related, and when one of them has a higher textual similarity against a particular query, the irrelevant file could be ranked higher.

Further research recognised the need for combining multiple analysis approaches on top of IR to support program comprehension (Gethers et al. 2011). To determine the starting points, like class and method names, in investigating relevant source code files for maintenance work, techniques combining dynamic (Wilde and Scully 1995) and static (Marcus et al. 2005) analysis have been exploited (Poshyvanyk et al. 2007; Eisenbarth et al. 2001; Le et al. 2015). However, requests for new non-existing features are unsuitable for dynamic analysis (Poshyvanyk et al. 2007). A large software project or one with a long history may require time-consuming analysis, making static approaches impracticable (Rao and Kak 2011).

1.1 Vocabulary of bug reports in source code files

In general, a typical BR document provides multiple fields where information pertaining to the reported issue may be described, such as a brief summary of the problem, a detailed description of the conditions observed, date of the observed behaviour, name of the files changed to resolve the reported condition. Recent empirical studies provide evidence that many terms used in BRs are also present in the source code files (Saha et al. 2013; Moreno et al. 2013). Such BR terms are an exact or partial match of program elements (i.e. class, method or variable names and comments) in at least one of the files affected by the BR, i.e. those files actually changed to address the BR.

Moreno et al. (2013) showed that (1) the BR documents share more terms with the corresponding affected files and (2) the shared terms were present in source file names. The authors evaluated 6 Open Source Software (OSS) projects, containing over 35K source files and 114 BRs, which were solved by changing 116 files. For each BR and source file combination (over 1 million), they discovered that on average 75% share between 1-13 terms, 22% share nothing and only 3% share more than 13 terms. Additionally, the study revealed that certain locations of a source code file, e.g. a file name instead of a method signature, may have only a few terms but all of them may contribute to the number of shared terms between a BR and its affected files. The authors concluded that the BRs have more terms in common with the affected files and the common terms are present in the names of those affected files.

Saha et al. (2013) claimed that although class names are typically a combination of 2–4 terms, they are present in more than 35% of the BR summary fields and 85% of the BR description fields of the OSS project AspectJ. Furthermore, the exact file name is present in more than 50% of the bug descriptions. They concluded that when the terms from these locations are compared during a search, the noise is reduced automatically due to reduction in search space. For example, in 27 AspectJ BRs, at least one of the file names of the fixed files was present as-is in the BR summary, whereas in 101 BRs at least one of the file name terms was present.

1.2 Our aim and contributions

Motivated by these insights, we aim to check if the occurrence of file names in BRs can be leveraged for IR-based bug localisation in Java programs. We restrict the scope to Java programs, where each file is a class or interface, in order to directly match the class and interface names mentioned in the BRs to the files retrieved by IR-based bug localisation.

Current state-of-the-art approaches for Java programs [BugLocator (Zhou et al. 2012), BRTracer (Wong et al. 2014), BLUiR (Saha et al. 2013), AmaLgam (Wang and Lo 2014), LearnToRank (Ye et al. 2014), BLIA (Youm et al. 2015) and Rahman et al. (2015)] rely on project history to improve the suggestion of relevant source files. In particular they use similar BRs and recently modified files. The rationale for the former is that if a new BR $\boldsymbol{x}$ is similar to a previously closed BR $\boldsymbol{y}$, the files affected by $\boldsymbol{y}$ may also be relevant to $\boldsymbol{x}$. The rationale for the latter is that recent changes to a file may have led to the reported bug. However, the observed improvements using the history information have been small.

We thus wonder whether file names mentioned in the BR descriptions can replace the contribution of historical information in achieving comparable performance and ask our first research question (RQ) as follows.

RQ1 *Can the occurrence of file names in BRs be leveraged to replace project history in achieving state-of-the-art IR-based bug localisation?*

If file name occurrence can’t be leveraged, we need to look more closely at the contribution of past history, in particular of considering similar bug reports, an approach introduced by BugLocator and adopted by others. So we ask in our second RQ:

RQ2 *What is the contribution of using similar bug reports?*

Furthermore, IR-based approaches to locating bugs use a base IR technique that is applied in a context-specific way or combined with bespoke heuristics. However, Saha et al. (2013) note that the exact variant of the underlying tf/idf (term frequency/inverse document frequency) model used may affect results. In particular they find that the off-the-shelf model they use in BLUiR already outperforms BugLocator, which introduced rVSM, a bespoke VSM variant. In our approach we also use an off-the-shelf VSM tool, different from the one used by BLUiR. In comparing our results to theirs we must therefore distinguish what is the contribution of file names, and what is the contribution of the IR model used. Thus we ask in our third RQ:

RQ3 *What is the overall contribution of the VSM variant adopted in our approach, and how does it perform compared to rVSM?*

Previous studies performed by Starke et al. (2009) and Sillito et al. (2008) reveal that text-based searches available in current IDEs are inadequate because they require search terms to be precisely specified, otherwise irrelevant or no results are returned. They highlighted that large search results returned by the IDE tools cause developers to analyse several files before performing bug-fixing tasks. Thus we wanted to know how developers perceive the search results of our tool, which presents a ranked list of candidate source code files that may be relevant for a BR at hand during software maintenance. Hence we ask our fourth RQ:

RQ4 *How does our approach perform with industrial applications and does it benefit developers?*

To address RQ1 we propose a novel approach and then evaluate it against existing approaches (Zhou et al. 2012; Saha et al. 2013; Moreno et al. 2014; Wong et al. 2014; Wang and Lo 2014; Ye et al. 2014) on the same datasets and with the same performance metrics. Like other approaches, ours scores each file against a given BR and then ranks the files in descending order of score, aiming for at least one of the files affected by the BR to be among the top-ranked ones, so that it can serve as an entry point to navigate the code and find the other affected files. As we shall see, our approach outperforms the existing ones in the majority of cases. In particular it succeeds in placing an affected file among the top-1, top-5 and top-10 files for 44, 69 and 76% of BRs, on average.

Our scoring scheme does not consider any historical information in the repository, which contributes to an *ab-initio* applicability of our approach, i.e. from the very first bug report submitted for the very first version. Moreover, our approach is efficient, because of the straightforward scoring, which doesn't require any further processing like dynamic code analysis to trace executed classes by re-running the scenarios described in the BRs.

To address RQ2, we compare the results of BugLocator and BRTracer using SimiScore (the similar bug score), and the results of BLUiR according to the literature, showing that SimiScore's contribution is not as high as suggested. From our experiments, we conclude that our approach localises many bugs without using similar bug fix information, which were only localised by BugLocator, BRTracer or BLUiR using similar bug information.

As for RQ3, through experiments we found that VSM is a crucial component to achieve the best performance for projects with a larger number of files that makes the use of term and document frequency more meaningful, but that in smaller projects its contribution is rather small. The Lucene VSM we chose performs in general better than the bespoke VSM of BugLocator and BRTracer.

We address RQ4 by conducting a user case study in 3 different companies with 4 developers. On average, our tool placed at least one affected file into the top-10 for 9 out 10 BRs. Developers stated that since most of the relevant files were positioned in the top-5, they were able to avoid the error prone tasks of browsing long result lists and performing repetitive search queries.

The rest of this paper is organised as follows. Section 2 describes the IR-based approaches against which we compare ours, and the datasets used to evaluate all of them. Section 3 describes our approach, detailing the scoring algorithm. Section 4 presents the results, addressing the first three research questions above. The user study is presented in Sect. 5. We discuss why the approach works and the threats to validity in Sect. 6. Finally, Sect. 7 presents concluding remarks.

2 Previous approaches

In bug localisation a query is a bug report, which is substantially different in structure and content from other text documents, thus requiring special techniques. Zhou et al. (2012) proposed an approach consisting of the four traditional IR steps (corpus creation, indexing, query construction, retrieval & ranking) but using a revised Vector Space Model (rVSM) to score each source code file against the given BR. In addition, each file gets a similarity score (SimiScore) based on whether the file was affected by one or more closed BRs similar to the given BR. Similarity between BRs is computed using VSM. The rVSM and similarity scores are each normalised to a value from 0 to 1 and combined linearly into a final score: $(1 - w)$*normalrVSM + w*normalSimiScore, where w is an empirically set weight.

The final score is then used to rank files from the highest to the lowest. The approach, implemented in a tool called BugLocator, was evaluated using over 3400 reports of closed bugs (see Table 1) and their known affected files from four OSS projects: the IDE tool Eclipse, the aspect-oriented programming library AspectJ, the GUI library SWT and the bar-code tool ZXing. Eclipse and AspectJ are well-known large scale applications used in many empirical research studies for evaluating various IR models (Manning et al. 2008). SWT is a subproject of Eclipse and ZXing is an Android project maintained by Google.

Table 1 Project artefacts

Project	Source files	Bug reports	Period	Used also by
AspectJ	6485	286	2002/07–2006/10	Zhou et al. (2012), Saha et al. (2013), Wong et al. (2014), Wang and Lo (2014) and Youm et al. (2015)
Eclipse	12863	3075	2004/10–2011/03	Zhou et al. (2012), Saha et al. (2013), Wong et al. (2014) and Wang and Lo (2014)
SWT	484	98	2004/10–2010/04	Zhou et al. (2012), Saha et al. (2013), Wong et al. (2014), Wang and Lo (2014), Youm et al. (2015) and Rahman et al. (2015)
ZXing	391	20	2010/03–2010/09	Zhou et al. (2012), Youm et al. (2015) and Rahman et al. (2015)
Tomcat	2038	1056	2002/07–2014/01	Ye et al. (2014)
ArgoUML	1685	91	2002/01–2006/07	Moreno et al. (2014)
Pillar1	4355	27	2012/03–2013/01	–
Pillar2	337	12	2010/05–2011/01	Dilshener and Wermelinger (2011)

For two of these projects w was set to 0.2 and for the other two $w = 0.3$. The performance of BugLocator was evaluated with 5 metrics: Top-1, Top-5, Top-10, MAP (mean average precision) and MRR (mean reciprocal rank). Top-N is the quantity (given as an absolute number or as a percentage) of bugs that are located with threshold N, which means that at least one affected file was placed among the first N ranked files. Across the four projects, BugLocator achieved a Top-10 of 60–80%, i.e. for each project at least 60% of its bugs were affected by at least one of the first 10 suggested files.

Wang et al. (2014) proposed a composite VSM approach that combines 5 variants of term frequency (tf) and 3 variants of inverse document frequency (idf)[1] to obtain 15 VSM variants. A final composite VSM is the weighted sum of all variants. A genetic algorithm (GA) was used with three-fold cross validation to find the weights that maximised the MAP and MRR. The approach was evaluated with 3 projects of the BugLocator dataset: AspectJ, Eclipse, and SWT. The composite approach, compared to the standard tf-idf, improved the MAP and MRR by at most 0.03 for AspectJ and SWT, but achieved no improvement for Eclipse.

2.1 Using stack trace and structure

It has been argued that existing approaches treat source code files as one whole unit and bug report (BR) as one plain text (Poshyvanyk et al. 2007; Saha et al. 2013; Ye et al. 2014; Kevic and Fritz 2014; Abebe et al. 2011). However, BRs may contain useful information like stack traces, code fragments, patches and recreation steps (Bettenburg et al. 2008). A study done by Schröter et al. (2010) explored the usefulness of stack trace (ST) information found in the BRs. They investigated 161,500 BRs in the Eclipse project bug repository and found that around 8% (12947/161500) BRs contained ST information. After manually linking 30% (3940/12947) of those closed BRs to their affected files, 60% (2321/3940) revealed that the file modified to solve the bug was one of the files listed in the ST.

Schröter et al. (2010) note that STs identified in the study contained up to 1024 file names and a median of 25 files. Out of 2321 BRs with a ST, in 40% the affected file was in the very first position of the ST and in 80% it was within the first 6 positions. It is not clear if non-project specific file names were ignored during the evaluation of the stack frames. Also, the authors discovered multiple stack traces in 32% (4206/12947) of the BRs with an ST. Investigating the affected files revealed that 70% of the time the affected file is found in the first trace and in 20% it is found on the second or third trace. Finally, the authors investigated whether including ST in BRs speeds up the development process and conclude that fixing bugs with ST information requires 2–26 days and without ST 4–32 days.

Moreno et al. (2014) presented LOBSTER, which leverages the ST information available in BRs to suggest relevant source code files. The authors argue that if 60% of the cases the ST contains at least one of the source code files that was changed to resolve the bug (Schröter et al. 2010), then in rest of the cases (40%) the changed source file should be a neighbouring one to those present in the ST. Based on this claim, the

[1] http://nlp.stanford.edu/IR-book/html/htmledition/tf-idf-weighting-1.html

approach first calculates a textual similarity score between the words extracted by the Lucene tool, which uses VSM, from bug reports and code files. Second, a structural similarity score is calculated between each file and the class names extracted from the stack trace information found in the bug report. If the file is not in the list of stack trace files then the application's call-graph information is used to check whether a neighbouring class's name occurs in the stack trace. Finally, both textual and structural similarity scores are combined to rank the files.

After evaluating the approach with only 155 bug reports containing a stack trace, the results reveal that 65% of the bugs were fixed in a file mentioned in the ST submitted in the BR. Out of 314 files fixed to resolve the reported bugs, 35% (109/314) were at a distance 1 from the files listed in the ST, 11% (36/314) at a distance of 2 and 8% (25/314) at a distance between 3 and 7. They observed that when parts of the code that need to be changed are not structurally related to the files listed in the ST then the effectiveness of the approach degraded. However, they conclude that overall 50% of the files to be changed were in the top-4 ranked files. Compared to standard VSM (Lucene), the approach achieved 53% better, 29% equal and 18% worse performance results.

Moreno et al. (2014) conclude that considering stack traces does improve the performance with respect to only using Lucene's VSM. Since our approach also leverages stack traces, we compare the performance of both tools using their dataset for ArgoUML, a UML diagraming tool (Table 1).

Wong et al. (2014) proposed BRTracer, which also leverages the ST information and performs segmentation of source files to reduce any noise due to varying size of the file lengths (Bettenburg et al. 2008). The method involves dividing each source file into sections, so called segments, and matching each segment to a BR to calculate a segment score and a length score. In addition, the ST is evaluated to calculate a ST boost score. Subsequently, for each source file, the segment with the highest score is taken as the relevant one for the BR and multiplied by the length score to derive a similarity score. Finally the ST score is added on top of the similarity score to get the overall score for a file.

The length score is calculated by revising the logistic function used in BugLocator (Zhou et al. 2012) with a beta factor to regulate how much favour is given to larger files. The ST boost score is calculated by evaluating the files listed in the stack trace as well as the files referenced via import statements, defined as the closely referred ones. The approach considers only the first 10 files listed in a ST to be relevant, which was first introduced by Schröter et al. (2010). Additionally, BugLocator's SimiScore function is utilised to calculate the similarity of the BR to the previously closed bugs and give favour to the files affected by those similar reports.

BRTracer was evaluated by measuring its performance against BugLocator with 3 of the same datasets used in that study to see whether the segmentation and ST makes a difference. Wong et al. (2014) concluded that the approach is able to significantly outperform the one introduced in BugLocator regardless of considering similar BRs or not. Authors claim that segmentation or ST analysis is an effective technique to boost bug localisation and both are compatible with the use of similar BRs. Since we also use the same datasets as BugLocator and leverage stack traces, we compare the performance of our tool against BRTracer to demonstrate the improvements gained and to confirm that ST analysis aids bug localisation.

2.2 Version history and other data sources

Nichols (2010) argues that utilising additional sources of information would greatly aid IR models to identify relations between BRs and source code files. One of the information sources available in well-maintained projects is the past bug details and to take advantage of this, the author proposes an approach that mines the past bug information automatically. The approach extracts semantic information from source code files (e.g. terms from identifiers and method names) to create searchable repositories. Also, one repository is augmented with information from previous BRs (e.g. title, description and affected files).

Experimenting with augmented and non-augmented search repositories revealed that search results from the repository augmented with up to 14 previous BRs were the same as those from the non-augmented one. The best performance was obtained when 26–30 previous bugs were used in the augmented repository. Nichols (2010) observed that there is no guarantee that considering previous bug information contributes towards improving the rank of a source file in the result list, and concludes that quality BRs, e.g. those with good descriptions, are more valuable than the number of past BRs included in the repository.

In their study about diversifying data sources to improve concept location, Ratanotayanon et al. (2010) ask if having more diverse data in the repository where project artefacts, e.g. source files, are indexed always produce better results. They investigated the effects of combining information from (1) change set comments added by developers when interacting with source version control systems, (2) BR details from issue tracking systems and (3) source code file dependency relations, i.e. the call graph.

They observed that (1) using change sets together with BRs produces the best results and (2) including the referencing files, i.e. the caller/callee information available in the call-graph, increased recall but deteriorated precision.

Ratanotayanon et al. (2010) claim that change sets provide vocabulary from the perspective of the problem domain because developers add to each commit descriptive comments which are more aligned with the terms used when performing a search task. However they argued that the success of utilising the change sets in search tasks is sensitive to the quality of these comments. Also, the authors proposed that when using a call-graph, the results be presented in ranked order based on a weighting scheme to compensate for any possible deterioration in precision due to false positives.

Wang and Lo (2014) proposed AmaLgam for suggesting relevant buggy source files by combining BugLocator's SimiScore and BLUiR's structured retrieval (Saha et al. 2013) into a single score using a weight factor, which is then combined (using a different weight) with a version history score that considers the number of bug fixing commits that touch a file in the past *k* days. Instead of considering all the previous BRs as in Sisman and Kak (2012), they only consider recent version history commits. They identify when the current bug was reported and compare the number of days/h between each past commit to assign a score to reflect the relevance of those previously modified files for the bug at hand. So if a file was changed 17 h ago, it gets a higher score then if it was changed 17 days ago.

The approach is evaluated in the same way as BugLocator and BLUiR, i.e. the number of BRs placed in top-N, for various values of *k*. AmaLgam matches or outperforms the other two in all indicators except one, the MAP for ZXing. Their evaluation shows that considering historical commits up to 15 days increased the performance, 15–20 days did not make much difference and considering up to 50 days deteriorated the performance. Thus they conclude that the most important part of the version history is in the commits of the last 15–20 days.

Wang et al. incorporated their composite VSM approach (Wang et al. 2014) into AmaLgam (Wang and Lo 2014) and compared the performance of AmaLgam with and without composite VSM. The MAP and MRR improved by 0.1 and 0.07 for AspectJ, by 0.04 and 0.03 for Eclipse, and the MAP improved by 0.01 for SWT. One of the shortcomings of the approach is that it takes time, as acknowledged by the authors: the training phase takes 3 h.

2.3 Combining multiple information sources

Saha et al. (2013) claim that dynamic approaches are more complicated, time consuming and expensive than static approaches where a recommendation system may be used, thus making the static approach more appealing. They argue that existing techniques treat source code as flat text ignoring the rich structure of source code file names, method names, comments, variables etc. The authors presented BLUiR, which leverages the structures inside a bug report and a source code file. To measure the similarity between a bug report and a file, the approach uses the Indri tf/idf model, which is based on BM25 (Okapi)[2] instead of VSM. It also incorporates structural information retrieval, where the textual content of each field in a bug report and each part of a source file are considered independently.

This involves dividing a bug report into summary and description fields, as well as dividing a source file into class names, method names, variable names, and comments. Indri computes a similarity score between each of the 2 fields of a bug report and each of the 4 parts of a source file. The 8 scores are summed into a final score to rank the files. One of the weaknesses of the approach is that it assumes all features are equally important and ignores the lexical gap between bug reports and source code files.

The results were evaluated using BugLocator's dataset and performance indicators. For all but one indicator for one project (ZXing's MAP), BLUiR matches or outperforms BugLocator, hinting that a different IR approach might compensate for the lack of history information, namely the previously closed similar bug reports. Subsequently, BLUiR incorporated BugLocator's SimiScore, which did further improve its performance.

Ye et al. (2014) claim that if a source file is recently changed than it may still contain defects and if a file is changed frequently than it may be more likely to have additional errors. They argued that there is a significant inherent mismatch between the descriptive natural language vocabulary of a BR and the vocabulary of a programming language found in the source code files. They defined a ranking model that combines six features measuring the relationship between bug reports and source code files, using a learning-to-rank (LtR) technique: (1) lexical similarity between bug reports

[2] http://nlp.stanford.edu/IR-book/html/htmledition/okapi-bm25-a-non-binary-model-1.html

and source code files; (2) API enriched lexical similarity, using API documentation of the libraries used by the source code; (3) collaborative filtering, using similar bug reports that got fixed previously; (4) bug fixing recency, i.e. time of last fix in terms of months; (5) bug fixing frequency, i.e. how often a file got fixed; (6) feature scaling, used to bring the score of all previous features into one scale. Their experimental evaluations show that the approach places a relevant file within the top-10 recommendations for over 70% of the bug reports of Tomcat (Table 1).

The source files are ranked based on the score obtained by evaluating each of the 6 features. Although improved results are obtained compared to existing tools, Ye et al. reported that in two datasets, AspectJ and Tomcat, existing tools also achieved very similar results. One of the reasons is that in AspectJ affected files were very frequently changed and in Tomcat the BRs had very detailed and long descriptions. Evaluation on performance of each feature shows that the best results were obtained by lexical similarity and considering previous BRs, so this leaves the question whether considering other features like version history commits as introduced in Wang and Lo (2014) is really worth the effort since other studies also report their effectiveness to be poor.

LtR approaches make use of machine learning in order to learn the rank of each document in response to a given query. Our simpler approach only uses the first of Ye et al. (2014) six features, lexical similarity, and yet provides better results on Tomcat, as we'll show.

Youm et al. (2015) introduced an approach where the scoring methods utilised in previous studies (Zhou et al. 2012; Saha et al. 2013; Wong et al. 2014; Wang and Lo 2014) are first calculated individually and then combined together by varying alpha and beta parameter values. The approach, implemented in a tool called BLIA, is compared against the performance of the other tools where the original methods were first introduced. For evaluation only the three smaller BugLocator datasets (i.e. excluding Eclipse) are used. Although BLIA improves the MAP and MRR values of BugLocator, BLUiR and BRTracer, it fails to outperform AmaLgam in AspectJ. The authors found that stack-trace analysis is the highest contributing factor among the analysed information for bug localisation.

Recently, Ye et al. (2016) used extra documents (tutorials, implementation guides, etc.) to obtain co-occurring words and thus be able to find files that have no word in common with the bug report. For each project, they train their approach on 2000 bug reports. They do not report top-N results, so it is unclear how well the approach performs in practice. Compared to their previous LtR approach, the use of extra documents improves the MAP and MRR by at most 0.03, except for the Eclipse JDT, with an improvement of 0.07.

In another recent study, Rahman et al. (2015) extended BugLocator by considering file fix frequency and file name matches. The file fix frequency score is calculated based on the number of times a ranked file is mentioned in the bug repository as changed to resolve another bug. The file name match score is based on the file names that appear in the bug report. The approach is evaluated with the SWT and ZXing projects from the BugLocator dataset and one of their own: Guava. The authors show improved MAP and MRR values over BugLocator's.

Independently of Rahman et al. (2015) (of which we became aware only recently), we decided to use file names because the study by Saha et al. (2013) shows that many bug reports contain the file names that need to be fixed. Rahman et al. (1) extract the file names from the bug report using very simple pattern matching techniques and (2) use a single constant value to boost a file's score when its name matches one of the extracted file names. On the contrary, our approach (1) uses a more sophisticated file matching regular expression pattern to extract file names from the bug report and (2) assigns varying values depending on the extracted file name's position in the bug report, as we will show later.

Uneno et al. (2016) proposed a new approach called Distributed REpresentation of Words based Bug Localization (DrewBL) which utilises a semantic Vector Space Model (sVSM). The idea is to concatenate the terms extracted from source code files and the stemmed words extracted from bug reports into a single vector space with a low dimension and high density. The approach combines the DrewBLScore with the scores obtained by running BugLocator and Bugspots,[3] which is based on bug-fixing history. The approach is tested on two OSS projects (Tomcat and Birt) from the LtR dataset (Ye et al. 2014) and the results reveal that DrewBL by itself performs substantially worse than the combined approach, which is still worse than LtR. The reason, as acknowledged, is that many irrelevant source code files are retrieved.

2.4 User studies

Sillito et al. (2008) conducted two different studies, one in a laboratory setting with 9 developers who were new to the code base and the other in an industrial setting with 16 developers who were already working with the code base. In both studies developers were observed performing change tasks to multiple source files within a fairly complex code base using modern tools. The findings reveal that text-based searches available in current IDEs are inadequate because they require search terms to be precisely specified, otherwise irrelevant or no results are returned. The study claims that developers repeatedly perform discovery tasks in a trial and error mode, which causes additional effort and often results in several failed attempts.

Starke et al. (2009) performed a study with 10 developers to find out how developers decide what to search for and what is relevant for the maintenance change task at hand. Participants were randomly assigned one of 2 closed bug descriptions selected from the Sub-Eclipse tool's issue repository and instructed to carry out search tasks in the Eclipse IDE using the available search tools. The findings highlight that formulating a search query is the most challenging task for the developers since Eclipse's search tools require the search terms to be precisely specified, otherwise no relevant results are returned. The authors also state that when many search results are returned, the developers tend to lose confidence in the query and decide to search again rather than investigate what was returned. They propose future research on tool support for developers, to provide more contextual information and to present results in a ranked order, grouped within the context provided by search tasks at hand.

[3] http://github.com/igrigorik/bugspots

Recently Kochhar et al. (2016) performed a study with practitioners about their expectation of automated fault localisation tools. The study explored several crucial parameters, such as trustworthiness, scalability and efficiency. Out of 386 responses, 30% rated fault localization as an "essential" research topic. The study further reveals that around 74% of respondents did not consider a fault localisation session to be successful if it requires developers to inspect more than 5 program elements and that 98% of developers indicated that inspecting more than 10 program elements is beyond their acceptability level. These findings show the importance of positioning relevant files in the top-10, especially in the top-5; otherwise the developers lose confidence.

Parnin and Orso (2011) studied users fixing 2 bugs with Tarantula, a tool that suggests a ranked list of code statements to fix a failed test. Users lost confidence if the ranked list was too long or had many false positives. Users didn't inspect the ranked statements in order, often skipping several ranks and going up and down the list. For some users, the authors manually changed the ranked list, moving one relevant statement from rank 83 to 16, and another from rank 7 to rank 35. There was no statistically significant change in how fast the users found the faulty code statements. This may be further indication that the top-5 ranks are the most important.

Xia et al. (2016) record the activity of 36 professionals debugging 16 bugs of 4 Java apps, each with 20+ KLOC. They divide professionals into 3 groups: one gets the buggy statements in positions 1–5, the other in positions 6–10, the other gets no ranked list. On average, the groups fix each bug in 11, 16 and 26 min, respectively. The difference is statistically significant, which shows the ranked list is useful. Developers mention that they look first at the top-5, although some still use all of the top-10 if they're not familiar with code. Some do an intersection of the list with where they think the bug is and only inspect those statements. Although the bug localisation is based on failed/successful tests and the ranked list contains code lines, not files, the study emphasises the importance of the top-5 and how developers use ranked lists.

As we shall present in Sect. 4, our approach equals or improves the top-5 metric on all analysed projects, and as we shall see in Sect. 5, our user study on IR-based localisation of buggy files confirms some of the above findings on test-based localisation of buggy code lines.

3 Our Approach

Each approach presented in the previous section incorporates an additional information source to improve results, as shown in Table 2. We list the six tools against which we evaluate our approach (ConCodeSe), by using the same datasets (Table 1) and metrics. Additionally, we used two financial applications, one open source, Pillar1, and one proprietary, due to confidentiality referred as Pillar2.

Both applications implement the financial regulations for credit and risk management defined by the Basel-II Accord (Basel 2006). Pillar1[4] is a client/server application

[4] http://www.pillarone.org

Table 2 Comparison of approaches

Approach	Underlying IR logic	Version history	Similar report	Structure	File name	Stack trace
BugLocator	rVSM	No	Yes	No	No	No
BRTracer	rVSM + segmentation	No	Yes	Yes	No	Yes
BLUiR	Indri	No	Yes	Yes	No	No
AmaLgam	Mixed	Yes	Yes	Yes	No	No
BLIA	rVSM + segmentation	Yes	Yes	Yes	No	Yes
Rahman	rVSM	Yes	Yes	No	Yes	No
LtR	VSM	Yes	Yes	Yes	No	No
LOBSTER	VSM	No	No	Yes	No	Yes
ConCodeSe	lexicalScore + VSM	No	No	Yes	Yes	Yes

developed in Java and Groovy by Munich Re (a re-insurance company). The BR documents are maintained with JIRA,[5] a public tracking tool.

Pillar2 is a web-based application developed in Java at our industrial partner's site and is not publicly available. It was in production for 9 years. The maintenance and further improvements were undertaken by five developers, including in the past the first author, none of them part of the initial team. The BR documents were maintained by a proprietary tracking tool.

As Table 1 shows, for both Pillar1 and Pillar2 we only have a small set of closed bug reports for which we also have the source code version on which they were reported. Neither application is maintained anymore.

3.1 Data processing

The two financial applications and the OSS projects from Table 1, consisting of source code files and BRs with their known affected files, identified as described in Zhou et al. (2012), Moreno et al. (2014) and Ye et al. (2014), were processed using our source code analysis tool called ConCodeSe (Contextual Code Search Engine), which we substantially improved from previous work (Dilshener and Wermelinger 2011) where only lexical similarity search was implemented. ConCodeSe utilises state of the art data extraction, persistence and search APIs (SQL, Lucene,[6] Hibernate[7]). Figure 1 illustrates the extraction, storage, search and analysis stages. In the top layer, the corpus creation and search services tasks are executed automatically.

The left hand side (1) represents the extraction of terms from the source code files and from the BRs. The middle part (2) shows the storage of the extracted terms. Finally, in the search stage (3), the search for the files affected by the BRs takes place.

[5] https://issuetracking.intuitive-collaboration.com

[6] http://lucene.apache.org/java/docs/index.html

[7] http://www.hibernate.org

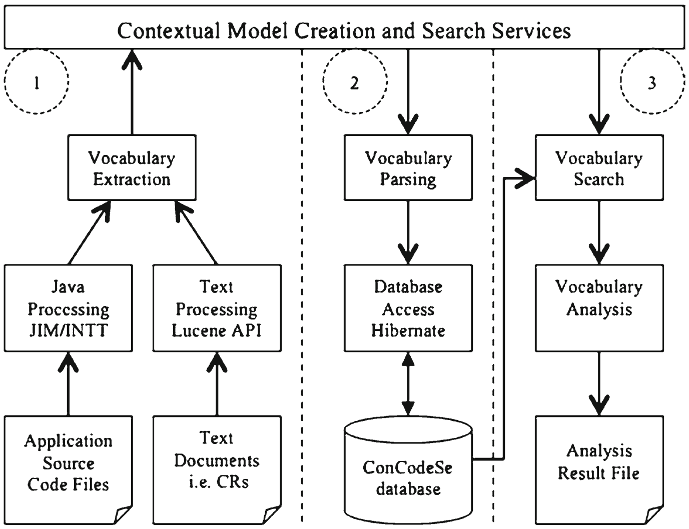


Fig. 1 ConCodeSe data extraction, storage and search

In the first stage, the Java code is parsed using the source code mining tool JIM[8] (Butler et al. 2010), which automates the extraction and analysis of identifiers from source files. It parses the code, extracts the identifiers and splits them into terms, using the INTT[9] tool (Butler et al. 2011) within JIM. INTT uses camel case, separators and other heuristics to split at ambiguous boundaries, like digits and lower case letters. The extracted information, i.e. identifier names, their tokenisation and source code location, is stored in Derby[10] relational database.

In the case of Pillar1, as it is developed in Groovy and Java, its BRs also refer to a mixture of both source code files. For BugLocator and BRTracer to process the Groovy source code files, we converted them into Java. Since Groovy is a dynamic language based upon Java with similar syntax and runs in the Java Virtual Machine, setting the file extensions as Java without modifying the file content was sufficient. Afterwards, manually inspecting the generated corpus confirmed the presence of the converted files with their associated terms extracted from the source code file identifiers.

Furthermore, the BRTracer tool, available at the online annex of Wong et al. (2014), is set to process only three datasets (AspectJ, Eclipse and SWT) as default. We were kindly granted access to its source code by its author (Chu Pan Wong) and with his assistance modified the main program to accept additional project artefact details, e.g. location of the source files and the BRs. Prior to running the modified tool using other datasets, we ran the modified version with the default projects and compared the results against those reported in its paper. This allowed us to verify that despite our modifications the results were as reported.

8 https://github.com/sjbutler/jim

9 https://github.com/sjbutler/intt

10 http://db.apache.org/derby

Also for the first stage, we developed a Java module to tokenise the text in the BRs into terms. The module reuses Lucene's StandardAnalyzer because it tokenises alphanumerics, acronyms, company names, email addresses, etc., using a JFlex-based lexical grammar. It also includes stop-word removal. We used a publicly available stop-words list[11] to filter them out. The extracted information is stored via the Hibernate persistence API into the same Derby database. We also use this module to extract terms (by tokenising words and removing stop-words) from Java source code comments prior to storing them into the database. During this stage, stemming is done using Lucene's implementation of Porter's stemmer (Porter 1997).

For the third stage of the process, we developed a Java module that (1) runs SQL queries to search for the occurrences of the BR terms in the source code files and (2) ranks all the files for each BR as explained in the next subsection. The ranked search results are saved in a spreadsheet for additional statistical analysis like computing the Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR) values (Boslaugh and Watters 2008).

MAP provides a single-figure measure of quality across recall levels. Among evaluation measures, MAP has been shown to have especially good discrimination and stability (Manning et al. 2008). The MAP is calculated as the sum of the average precision value for each BR divided by the number of BRs for a given project. The average precision (AP) for a BR is the mean of the precision values obtained for all affected files listed in the result set and computed as in Eq. 1. Then the MAP for a set of queries is the mean of the average precision values for all queries, which is calculated as in Eq. 2.

$$AP(Relevant) = \frac{\sum_{r \in Relevant} Precision(Rank(r))}{|Relevant|} \tag{1}$$

$$MAP(Queries) = \frac{\sum_{q \in Queries} AP(Relevant(q))}{|Queries|} \tag{2}$$

MAP is considered to be optimal for ranked results when the possible outcomes of a query are 5 or more (Lawrie 2012). As Table 3 shows, very few BRs have at least 5 affected files, but we still use the MAP for comparison with previous approaches. As we will show, for all eight projects, including ZXing, we achieve the best MAP.

MRR is a metric for evaluating a process that produces a list of possible responses to a query (Voorhees 2001). The reciprocal rank $RR(q)$ for a query q is the inverse rank of the first relevant document found and computed as in Eq. 3. Then the MRR is the average of the reciprocal ranks for the results of a set of queries calculated as in Eq. 4.

$$RR(q) = \begin{cases} if\, q\ retrieves\ no & \\ relevant\ documents & 0 \\ otherwise & \frac{1}{TopRank(q)} \end{cases} \tag{3}$$

11 http://norm.al/2009/04/14/list-of-english-stop-words/

$$MRR(Queries) = \frac{\sum_{q \in Queries} RR(q)}{|Queries|} \tag{4}$$

On the other hand, MRR is known to measure the performance of ranked results better when the outcome of a query is less than 5 and best when just 1 (Lawrie 2012), which is the majority of cases for the datasets under study (Table 3). The higher the MRR value, the better the bug localisation performance. We will show that our approach improves, sometimes substantially, the MRR for all eight projects.

3.2 Ranking files

Given a BR and a file, our approach computes two kinds of scores for the file: a lexical similarity score (Algorithm 2, explained later) and a probabilistic score given by VSM, as implemented by Lucene.

The two scorings are done with four search types, each using a different set of terms indexed from the BR and the file:

1. Full terms from the BR and from the file's code.
2. Full terms from the BR and the file's code and comments.
3. Stemmed terms from the BR and the file's code.
4. Stemmed terms from the BR, the file's code and comments.

For each of the 8 scorings, all files are ranked in descending order. Files with the same score are ordered alphabetically, e.g. if files F and G have the same score, F will be ranked above G. Then, for each file we take the best of its 8 ranks. If two files have the same best rank, then we compare the next best rank, and if that is tied, then their 3rd best rank etc. For example, if file B's two best ranks (of the 8 ranks each file has) is 1 and 1, and if file A's two best ranks are 1 and 2, B will be ranked overall before A.

The rationale for the four search types is that during experiments, we noticed that when a file could not be ranked among the top-10 using the full terms and the code, it was often enough to use associated comments and/or stemming. As shown in Table 4, for SWT's BR #100040, the affected Menu.java file had a low rank (484th) using the first type of search. When the search includes comments, stemming or both, it is

Table 3 Affected files per BR

Project	BRs with only 1 file	BRs with 2–4 files	BRs with $\geq$ 5 files
AspectJ	71	149	66
Eclipse	1525	1066	484
SWT	59	32	7
ZXing	14	5	1
Tomcat	690	281	85
ArgoUML	42	32	17
Pillar1	10	7	10
Pillar2	0	1	11

Table 4 Ranking achieved by all search types in SWT

BR #	Affected Java file	BugLocator	BRTracer	ConCodeSe	1: Full/code	2: Full/all	3: Stem/code	4: Stem/all
100040	Menu	20	65	**2**	484	3	29	**2**
79107	Combo	6	8	**3**	26	29	**3**	8
84911	FileDialog	6	6	**5**	**5**	39	6	56
92757	StyledText Listener	87	74	**3**	4	**3**	75	72

The best rank of each file is in bold

ranked at 3rd, 29th and 2nd place respectively. The latter is the best rank for this file and thus returned by ConCodeSe (4th column of the table).

There are cases when using comments or stemming could deteriorate the ranking, for example because it helps irrelevant files to match more terms with a BR and thus push affected classes down the rankings. For example, in BR #92757, the affected file StyledTextListener.java is ranked much lower (75th and 72nd) when stemming is applied. However, by taking the best of the ranks, ConCodeSe is immune to such variations.

The lexical similarity scoring and ranking is done by function *searchAndRankFiles* (Algorithm 1), which takes as arguments a BR and the project's files and returns an ordered list of files. The function is called 4 times, for each search type listed previously, and goes through the following steps for each source code file.

1. Check if the file's name matches one of the words in key positions (KP) of the BR's summary, and assign a score accordingly (Sect. 3.2.1).
2. If no score was assigned and if a stack trace (ST) is available, check if the file's name matches one of the file names listed in the ST and assign a score accordingly (Sect. 3.2.2).
3. If there is still no score then assign a score based on the occurrence of the search terms, i.e. the BR text terms (TT) in the file (Sect. 3.2.3).

Once all the files are scored against a BR, the list is sorted in descending order, where the files with higher scores are ranked at the top. Ties are broken by alphabetical order.

Algorithm 1 *searchAndRankFiles*

```
input: files: List<File>, br: BR // one bug report
output: ranked: List<File>
foreach file in files
  file.score := scoreWithKeyPositionWord(file.name, br.summary) // KP Score
  if file.score = 0 and br.stackTrace exists then
    file.score := scoreWithStackTrace(file.name, br.stackTrace) // ST Score
  endif
  if file.score = 0 then
    file.score := scoreWithFileTerms(file, br) // TT Score
  endif
end for
return files in descending order by score
```

Table 5 Sample of words in key positions

BR#	Summary	Position
79268	Program API does not work with GNOME 2.8 (libgnomevfs-WARNING)	First
78559	[consistency] Slider fires two selection events before mouse down	Second
92341	DBR - Add SWT.VIRTUAL style to Tree widget	Penultimate
100040	Slow down between 3.1 RC1 and N20050602 due to change to ImageList	Last

3.2.1 Scoring with key positions (KP score)

By manually analysing all SWT and AspectJ BRs and 50 random Eclipse BRs, i.e. $(98 + 286 + 50)/4665 = 9.3\%$ of all BRs (Table 1), we found that the word in the first, second, penultimate or last position of the BR summary may correspond to the affected file name. For example, Table 5 shows that in the summary sentence of BR #79268, the first word is already the name of the affected source file, i.e. Program.java.

Overall, our manual analysis of SWT revealed that in 42% (42/98) of the BRs the first word and in 15% (15/98) of the BRs the last word of the summary sentence was the affected source file. We found similar patterns in AspectJ: 28% (81/286) as the first word and 5% (15/286) as the last word. The frequency for the second and penultimate words being the affected file was 4 and 11% respectively.

We also noticed that some KP words come with method and package names in the BR summary, e.g. Class.method() or package.subpackage.Class.method(). They hence required parsing using regular expressions. Based on these findings, we assign a high score to a source file when its name matches the words in the above described four key positions of the BR summary sentence. The earlier the file name occurs, the higher the score: the word in first position gets a score of 10, the second 8, the penultimate 6 and the last 4. Note that the key positions are scored in a different order (1st, 2nd, penultimate, last) from their frequency (1st, penultimate, last, 2nd) in AspectJ, because while experimenting with different score values for SWT and AspectJ we found the ‘natural’ order to be more effective. Disregarding other positions in the summary prevents non-affected files that occur in those other positions from getting a high KP score and thus a higher ranking.

3.2.2 Scoring with stack traces (ST score)

Stack traces list the files that were executed when an error condition occurs. During manual analysis of the same BRs as for KP we found several included an ST in the description field (see Table 6).

We found that especially for NullPointerException, the affected file was often the first one listed in the stack trace. For other exceptions such as UnsupportedOperationException or IllegalStateException however, the affected file was likely the second or the fourth in the stack trace.

Table 6 Stack trace information in BRs

Project	# of BRs	# of BRs with Stack Traces	% of BRs with Stack Traces
AspectJ	286	67	23
Eclipse	3075	435	14
SWT	98	4	4
ZXing	20	0	0
Tomcat	1056	83	8
ArgoUml	91	5	5
Pillar1	27	1	4
Pillar2	12	0	0

We first use regular expressions (see Fig. 2) to extract from the ST the application-only source files, i.e. excluding third party and Java library files, and then put them into a list in the order in which they appeared in the trace. We score a file if its name matches one of the first four files occurring in the list. The file in first position gets a score of 9, the second 7, the third 5 and the fourth 3.

3.2.3 Scoring with text terms (TT score)

We assign a score to the source file based on where the BR's terms (without duplicates) occur in the file. If a BR term occurs in the file name this results in a higher score. Each occurrence of each BR term in the file increments slightly the score, as shown in function *scoreWithFileTerms* (Algorithm 2). As explained before, the BR's and the file's terms depend on whether stemming and/or comments are considered.

Again, the file names are treated as the most important elements and are assigned the highest score. When a query term is identical to a file name, it is considered a full match (no further matches are sought for the file) and a relatively high score (adding 2) is assigned. The occurrence of the query term in the file name is considered to be more important (0.025) than in the terms extracted from identifiers, method signatures or comments (0.0125).

The TT score values were chosen by applying the approach on a small sized training dataset, consisting of randomly selected 51 BRs from SWT and 50 BRs from AspectJ, i.e. $(50 + 51)/4665 = 2.17\%$ of all BRs, and making adjustments to the scores in order to tune the results for the training dataset.

3.2.4 Rationale behind the scoring values

As mentioned in the previous subsections, values for the KP scoring (10, 8, 6, 4), ST scoring (9, 7, 5, 3) and TT scoring (2, 0.025, 0.0125) were obtained heuristically,

```
at [^java. | ^sun.]([packageName]*).  // exclude java packages
([className]*).($[innerClassName]*)?. // consider any inner class
([methodName]*) (  ((([fileName]*).java):
[lineNumber])*? (Unknown Source)? (Native Method)? )
```

Fig. 2 Search pattern for stack trace

Algorithm 2 *scoreWithFileTerms*

```
input: file: File, br: BR // one bug report
output: score: float = 0
foreach query_term in br.terms
 if (query_term = file.name)
  score := score + 2
  return score
 endif
 if (file.name contains query_term) then
  score := score + 0.025
 else
  foreach doc_term in file.terms
   if (doc_term = query_term) then
    score := score + 0.0125
   endif
  endfor
 endif
endfor
return score
```

whilst reflecting the importance of the summary, the stack trace and the description, in this order, and of the different parts of each. The scoring values are weights that give more importance to certain positions in the summary and stack trace, or to certain kinds of words (class names). As such, they are in essence not different from other weights used by other authors, which they also obtained heuristically. For example, Hill et al. (2007) assigned weights 2.5 and 0.5 to terms based on whether they match those extracted from the file name or method name, respectively, and Uneno et al. (2016) summed the scores of three tools adjusted by weights 0.5, 0.3, and 0.1 or 1.0, based on manual experiments and observations. Methodologically, our approach therefore does not deviate from established practice.

Some authors automatically determine the best weight values, e.g. by using machine learning. However, the optimal weights obtained from a particular collection of projects are not necessarily the best weights for other projects. For example, in Tomcat the bug reports have very detailed and long descriptions but in Pillar2 they are tersely written. Optimising weights (via machine learning or some other technique) must therefore be done separately for each project, and only for projects that have sufficient history, i.e. sufficient past BRs that can be used as training set.

We are not interested in a tailored approach that provides the best results for the 8 projects at hand. Rather, we aim to see whether using the least information possible (the bug report to be localised and the source code files on which the bug was reported) and using a light-weight IR approach with fixed, not necessarily optimal, weights, we can achieve similar performance to other approaches that use more data sources (similar past bug reports, past code versions) or more complicated methods. As we will show in Sect. 4, we actually surpass the performance of other approaches. Our results can thus form a new baseline to compare to, and our simple minimalistic approach can serve as a ‘trampoline’ for others to improve on our baseline, e.g. by modifying Algorithm 1 or by adding further heuristics and techniques, like project history and machine learning.

4 Evaluation of the results

In this section we address the research questions. Since they ask for the effects of various scoring components, we had to run ConCodeSe, BugLocator and BRTracer (the other tools were unavailable) in different 'modes', e.g. with and without stack trace information or with and without SimiScore (similar bug reports), to observe the effect on the ranking of individual files. We ran BugLocator and BRTracer without SimiScore by setting the alpha parameter to zero, as described in Zhou et al. (2012) and Wong et al. (2014). We also ran both tools with SimiScore, by setting alpha to the value reported in the corresponding paper. We confirmed that we obtained the same top-N, MAP and MRR results as reported in the papers. This reassured us we were using the same tool versions, datasets and alpha values as the authors had, and that the results reported in the rest of this section for BugLocator and BRTracer are correct.

As we derived our heuristics by manually analysing the BRs of AspectJ and SWT, and some from Eclipse, to avoid bias, we evaluated our approach using additional OSS and industrial projects: ArgoUML, Tomcat, ZXing, Pillar1 and Pillar2. As already described in Sect. 2, all but the last two projects were also used by the approaches we compare our tool's performance against.

4.1 RQ1: scoring with file names in BRs

As described throughout Section 3, our lexical similarity scoring mainly considers whether the name of the file being scored occurs in the BR, giving more importance to certain positions in the summary or in the description's stack trace. The rationale is of course that a file explicitly mentioned in the BR is likely to require changes to fix the bug.

The first research question asks whether such an approach, although seemingly sensible, is effective. To answer the question we compare our results to those of BugLocator, BRTracer, BLUiR, AmaLgam, LtR, BLIA and Rahman using the same 5 metrics (Top-1, Top-5, Top-10, MAP, MRR) and for LOBSTER using only the MAP and MRR metrics (as given in their paper). We look at the separate and combined effect of using file names for scoring.

4.1.1 Scoring with words in key positions (KP score)

To see the impact of considering occurrences of the file name in certain positions of the BR summary, we ran ConCodeSe with and without assigning a KP score, whilst keeping all the rest as described in Sect. 3. Table 7 shows how evaluating the words in key positions improved results for the affected classes of the BRs given in Table 5. In the cases of BugLocator and BRTracer, we obtained the ranks by running their tool, and obtained the ones for BLUiR from their published results.

As the table illustrates, in some cases (like for Program.java) the summary scoring can make the difference between the file not even making into the top-10 or making into the top-5. In other cases, the change in ranking is small but can be significant,

Table 7 Results with and without considering key positions

SWT BR#	Affected file	BugLocator	BRTracer	BLUiR	ConCodeSe	
					Without	With
79268	Program.java	20	10	–	19	2
78559	Slider.java	2	4	1	5	1
92341	Tree.java	1	1	5	4	2
100040	ImageList.java	7	1	9	2	1

making the affected file become the top ranked, which is always the most desirable situation, as the developer will not have to inspect any irrelevant files.

To have an overall view, we also ran ConCodeSe using just KP and TT scores together (KP+TT) against only using TT score, i.e. in both runs ST and VSM scoring were not used. Table 8 shows that compared to TT alone, KP+TT provides an advantage in positioning files of a BR in the top-1 for SWT and ZXing, and in the top-5 for AspectJ, Eclipse and Tomcat. On the contrary, in the cases of Pillar1 and Pillar2 using KP+TT score did not make a difference and the results remained the same as the TT score in all top-N categories. Further analysis revealed the reason: in Pillar2 the BR summaries do not contain source code file names and in Pillar1 only 3 BRs contain file names in their summaries, but they are not the files changed to resolve the reported bug and the TT score of the relevant files is higher.

Overall 46–86% of BRs can be located by just assigning a high score to file names in certain positions of the BR summary, confirming the studies cited in the introduction that found file names mentioned in a large percentage of BRs (Saha et al. 2013; Schröter et al. 2010). The file name occurrences in other places of the BR are also scored by comparing BR and file terms in function *scoreWithFileTerms* (see Algorithm 2), but irrelevant files that match several terms may accumulate a large score that pushes the affected classes down the ranking.

4.1.2 Scoring with stack trace information (ST score)

To see the impact of considering file names occurring in a stack trace, if it exists, we ran ConCodeSe with and without assigning an ST score, but again leaving all the rest unchanged, i.e. using key position (KP) and text terms (TT) scoring. Table 9 shows results for some affected classes obtained by BugLocator, BRTracer and BLUiR.

Again, the rank differences can be small but significant, moving a file from top-10 to top-5 (ResolvedMemberImpl.java) or from top-5 to top-1 (EclipseSourceType.java). In some cases (ReferenceType.java) the file goes from not being in the top-10 to being in the top-1, even if it is in the lowest scoring fourth position in the stack.

Table 8 also shows the effect of running ConCodeSe just with and TT scores together (ST+TT) against only using TT score, i.e. without KP and VSM scoring, except for ZXing and Pillar2, which don't have any stack traces in their BRs (Table 6). ST+TT scoring provides significant advantage over the TT score alone in positioning affected files of a BR in top-1. In particular for AspectJ, Eclipse, Tomcat and ArgoUML, ST

Table 8 Key position versus stack trace versus text terms

Project	Scoring	Top-1(%)	Top-5 (%)	Top-10 (%)	MAP	MRR
AspectJ	KP+TT only	12.9	43.0	59.1	0.17	0.33
	ST+TT only	**21.7**	**45.1**	**59.4**	**0.20**	**0.40**
	TT only	13.6	43.7	59.1	0.17	0.34
Eclipse	KP+TT only	19.5	35.2	48.0	0.21	0.32
	ST+TT only	**19.9**	**35.9**	**48.0**	**0.21**	**0.33**
	TT only	18.3	34.7	48.0	0.20	0.32
SWT	**KP+TT only**	**62.2**	**79.6**	**89.8**	**0.60**	**0.81**
	ST+TT only	43.9	76.5	88.8	0.50	0.70
	TT only	42.9	76.5	88.8	0.50	0.69
ZXing	**KP+TT only**	**40.0**	**65.0**	**80.0**	**0.46**	**0.53**
	ST+TT only	N/A	N/A	N/A	N/A	N/A
	TT only	25.0	60.0	75.0	0.38	0.42
Tomcat	**KP+TT only**	**36.2**	**56.3**	**64.1**	**0.39**	**0.49**
	ST+TT only	34.8	56.3	64.3	0.39	0.49
	TT only	34.0	55.7	64.0	0.38	0.48
ArgoUML	KP+TT only	12.1	48.4	56.0	0.19	0.32
	ST+TT only	**13.2**	**48.4**	**56.0**	**0.20**	**0.33**
	TT only	12.1	48.4	56.0	0.19	0.32
Pillar1	KP+TT only	7.4	33.3	40.7	0.10	0.36
	ST+TT only	7.4	33.3	40.7	0.10	0.36
	TT only	7.4	33.3	40.7	0.10	0.36
Pillar2	KP+TT only	25.0	66.7	75.0	0.20	0.71
	ST+TT only	25.0	66.7	75.0	0.20	0.71
	TT only	25.0	66.7	75.0	0.20	0.71

The best scoring combination for each project is in bold

scoring places more BRs in all top-N categories, indicating that giving file names found in stack trace a higher score contributes to improving the performance of the results, which is also in line with the findings of previous studies (Schröter et al. 2010; Moreno et al. 2014; Wong et al. 2014).

Note that there is no significant difference between ST+TT and TT scoring for SWT and ArgoUML. Only 4 of SWT's BRs have a stack trace and it happens that

Table 9 Results with and without stack trace information

AspectJ BR#	Exception description	Affected class	Stack pos.	BugLocator	BRTracer	BLUiR	ConCodeSe	
							Without	With
138143	NullPointerException	EclipseSourceType	1st	1	2	5	5	1
158624	UnsupportedOperation	ResolvedMemberImpl	2nd	16	6	56	7	3
153490	IllegalStateException	ReferenceType	4th	122	3	74	11	1

Table 10 Key word and stack trace scoring on versus off

Project	KP+ST	Top-1 (%)	Top-5 (%)	Top-10 (%)	MAP	MRR
AspectJ	**On**	**42.3**	**68.2**	**78.3**	**0.32**	**0.67**
	Off	35.0	67.1	78.3	0.30	0.62
Eclipse	**On**	**37.6**	**61.2**	**69.9**	**0.37**	**0.57**
	Off	34.6	59.6	69.7	0.35	0.55
SWT	**On**	**72.4**	**89.8**	**92.9**	**0.68**	**0.94**
	Off	59.2	88.8	92.9	0.62	0.85
ZXing	**On**	**55.0**	**75.0**	**80.0**	**0.55**	**0.68**
	Off	35.0	70.0	80.0	0.45	0.52
Tomcat	**On**	**51.5**	**69.2**	**75.4**	**0.52**	**0.66**
	Off	49.1	68.6	75.3	0.51	0.65
ArgoUML	**On**	**31.9**	**61.5**	**65.9**	**0.30**	**0.55**
	Off	31.9	61.5	65.9	0.30	0.55
Pillar1	**On**	**29.6**	**59.3**	**63.0**	**0.22**	**0.69**
	Off	29.6	59.3	63.0	0.22	0.69
Pillar2	**On**	**33.3**	**66.7**	**83.3**	**0.26**	**0.92**
	Off	33.3	66.7	83.3	0.26	0.92

The results with scoring turned on are in bold

in those cases the lower TT score value of 2 for the files occurring in the stack trace is still enough to rank them highly. For ArgoUML, 5 bug reports contain stack trace information and using ST+TT scoring adds only one more bug report to the top-1 compared to the other two scoring variations. The small difference is due to the relevant file for the other 4 bug reports not being among the ones listed in the stack trace or being listed at a position after the 4th. Since ST scoring only considers the first 4 files, in that case the affected file gets a TT score, because its name occurs in the BR description.

Again, for Pillar1 and Pillar2 using ST+TT scoring alone did not make a difference and the results remained constant in all top-N categories. None of the Pillar2 BRs contains stack traces and in the case of Pillar1 only 1 BR description contains stack traces but the relevant file is listed after the 4th position and gets a ST score of zero (Sect. 3.2.2).

We end the analysis of the contributions of positional scoring of file names in BR summaries (KP) and stack trace (ST) with Table 10, which shows the combined rank 'boosting' effect of positional scoring, i.e. using KP and ST scoring together vs not using it. For example in the case of the SWT project, using summary and stack trace scoring places an affected source file in the top-1 for 72% of the BRs compared to the base case (TT score) of 59%. This performance advantage remains noticeable high for all the projects except for ArgoUML, Pillar1 and Pillar2 due to the stated reasons.

Table 11 KP and ST scoring variations

Project	Approach	Top-1 (%)	Top-5 (%)	Top-10 (%)	MAP	MRR
Aspectj	**halved**	**42.0**	**68.5**	**78.3**	**0.33**	**0.67**
	reversed	34.6	65.0	77.6	0.30	0.61
	uniform	37.4	68.2	78.3	0.31	0.64
	close2base	41.6	68.2	78.3	0.33	0.67
Eclipse	halved	37.2	61.1	69.8	0.37	0.57
	reversed	35.7	61.2	69.9	0.36	0.56
	uniform	36.6	61.2	69.9	0.36	0.56
	close2base	**37.8**	**67.8**	**78.3**	**0.36**	**0.57**
SWT	halved	71.4	89.8	92.9	0.68	0.93
	reversed	69.4	88.8	92.9	0.66	0.92
	uniform	71.4	89.8	92.9	0.68	0.94
	close2base	**72.4**	**89.8**	**92.9**	**0.68**	**0.94**
Tomcat	halved	50.9	68.8	75.5	0.52	0.65
	reversed	50.5	68.8	75.5	0.52	0.65
	uniform	51.0	69.2	75.4	0.52	0.66
	close2base	**51.7**	**69.1**	**75.4**	**0.52**	**0.66**
ArgoUML	**any**	31.9	61.5	65.9	0.30	0.55
ZXing	**any**	55.0	75.0	80.0	0.51	0.63
Pillar1	**any**	29.6	59.3	63.0	0.22	0.69
Pillar2	**any**	33.3	66.7	83.3	0.26	0.92

The best variation for each project is in bold

4.1.3 Variations of score values

The KP and ST scores are computed in very simple ways, which may affect the performance of the outcome. Variations to those scores may or may not further enhance the performance. We experimented further to evaluate the effects of our scoring mechanism by assigning different scores to the four positions in the summary and in the stack trace. Table 11 shows the results obtained after performing the following changes:

1. The scores were halved, e.g. 10, 8, 6, 4 became 5, 4, 3, 2.
2. The scores were reversed, i.e. 4, 6, 8, 10 and 3, 5, 7, 9.
3. All 8 positions (4 in the BR summary and 4 in the ST) have a uniform score of 10.
4. The scores were made closer to those of TT (close2base):
   (a) for the summary positions: 3.00, 2.75, 2.50, 2.25
   (b) for the stack positions: 2.75, 2.50, 2.25, 2.00

Halving and close2base keeps the order of each set of 4 positions, and the results obtained are similar to those obtained with the original score values, 10, 8, 6, 4 for summary positions and 9, 7, 5, 3 for stack trace positions. The reversed and uniform scoring break that order and led to the worst results. This confirms that the relative importance of the various positions (especially the first position) found through inspection of SWT and AspectJ applies to most projects.

In the cases of ArgoUML, Pillar1 and Pillar2 changing the KP and ST scoring doesn't make a difference because the 8 summary and stack trace positions do not play a role in those projects (Table 10). In the case of ZXing neither of the scoring variations made a difference due to the small number of BRs. It made little difference to SWT, which has only a few more BRs than ArgoUML.

Looking closer at Table 11, we note that in cases of the Eclipse, SWT and Tomcat projects, close2base places more BRs in top-1 than any other variation. Investigating more closely, we found that one additional BR for SWT and two for Tomcat are placed in top-1. In the case of SWT, the only affected source code file, Spinner.java for BR #96053, achieved a TT score of 2.56 by function *scoreWithFileNames* (see Algorithm 2) and is ranked in the 2nd position whereas the file Text.java achieved a KP score of 4 and is ranked 1st. Analysing further revealed that the last word "text" in the BR summary sentence matches the file name, thus assigning a high KP score value of 4 to the file Text.java. However, when close2base scores are used, the KP score value for the last word position is set to 2.25 (see point 4a in the variations list above), which is lower than the TT score (2.56), thus ranking Spinner.java as 1st and Text.java as 2nd. Similar patterns were discovered in Eclipse and Tomcat as well.

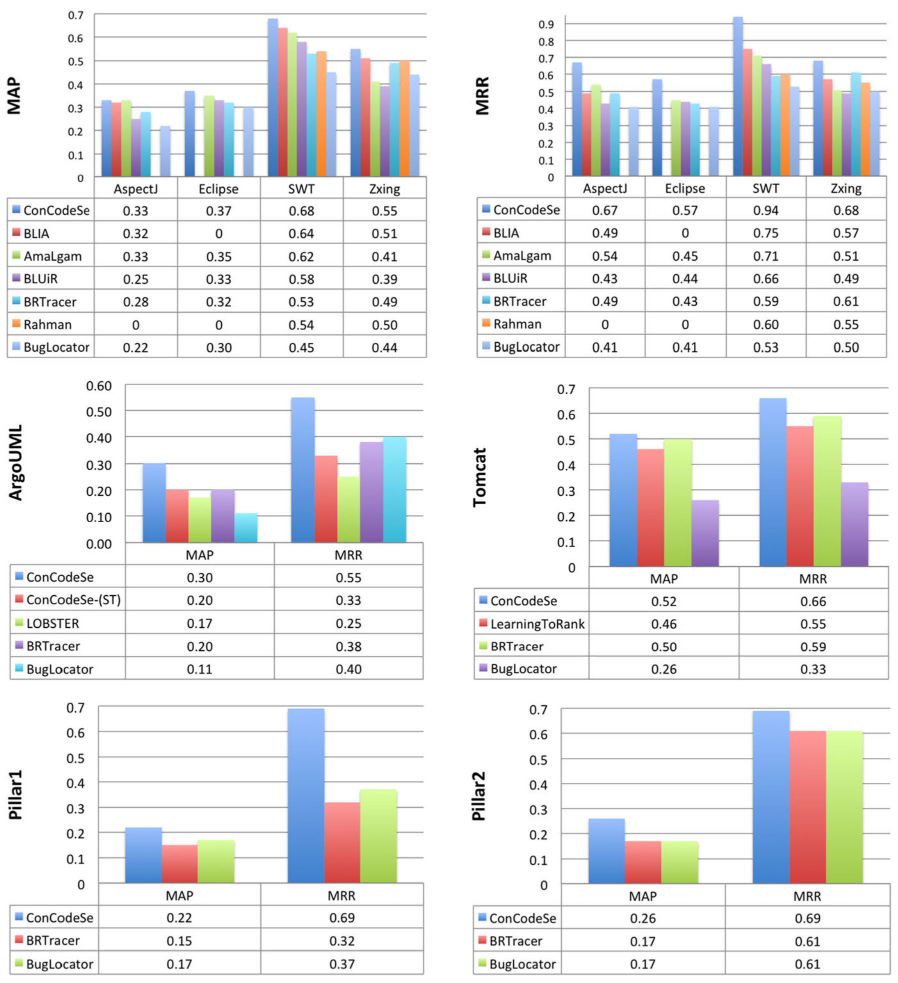
Comparing Table 11 to the results achieved by other approaches (Fig. 4 in the next subsection), we note that the halved and close2base variations outperform the other approaches in most cases, showing that the key improvement is the higher and differentiated scoring of the 4 positions in the summary and stack trace, independently of the exact score values.

To sum up, the four systematic transformations of the score values and the better performance of the halved and close2base transformations provide evidence that the absolute values is not the main point but rather ther relative values. Moreover, the heuristics (the more likely occurrence of relevant file names in certain positions) were based on the analysis of only 10% of the bug reports, far less than the typical training data sample used in k-fold cross-validation. Especially for large projects like Eclipse, with many people reporting bugs, one can reasonably expect that many BRs will deviate from the sample. The similar behaviour of the variants across all projects and all BRs (close2base and halved are better than uniform, which is better than reversed) therefore provides reassurance that the chosen values capture well the heuristics and that the heuristics are valid beyond the small sample size used to obtain them.

4.1.4 Overall results

Finally, we compare the performance of ConCodeSe against the state-of-the-art tools using their datasets and metrics (Figs. 3, 4). As mentioned before, we were only able to obtain BugLocator and BRTracer,[12] which meant that for the other approaches we could only refer to the published results for the datasets they used. This means we could compare our Pillar1 and Pillar2 results only against those two approaches and couldn't

[12] Although BLIA is available online, we were unable to run it on datasets other than the ones used by its authors (AspectJ, SWT and ZXing).



MAP	AspectJ	Eclipse	SWT	Zxing
ConCodeSe	0.33	0.37	0.68	0.55
BLIA	0.32	0	0.64	0.51
AmaLgam	0.33	0.35	0.62	0.41
BLUiR	0.25	0.33	0.58	0.39
BRTracer	0.28	0.32	0.53	0.49
Rahman	0	0	0.54	0.50
BugLocator	0.22	0.30	0.45	0.44

MRR	AspectJ	Eclipse	SWT	Zxing
ConCodeSe	0.67	0.57	0.94	0.68
BLIA	0.49	0	0.75	0.57
AmaLgam	0.54	0.45	0.71	0.51
BLUiR	0.43	0.44	0.66	0.49
BRTracer	0.49	0.43	0.59	0.61
Rahman	0	0	0.60	0.55
BugLocator	0.41	0.41	0.53	0.50

ArgoUML	MAP	MRR
ConCodeSe	0.30	0.55
ConCodeSe-(ST)	0.20	0.33
LOBSTER	0.17	0.25
BRTracer	0.20	0.38
BugLocator	0.11	0.40

Tomcat	MAP	MRR
ConCodeSe	0.52	0.66
LearningToRank	0.46	0.55
BRTracer	0.50	0.59
BugLocator	0.26	0.33

Pillar1	MAP	MRR
ConCodeSe	0.22	0.69
BRTracer	0.15	0.32
BugLocator	0.17	0.37

Pillar2	MAP	MRR
ConCodeSe	0.26	0.69
BRTracer	0.17	0.61
BugLocator	0.17	0.61

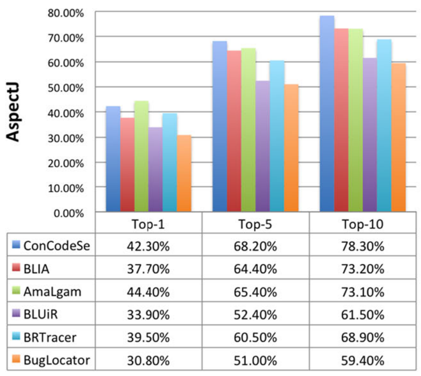
Fig. 3 MAP and MRR values of the tools

for example run BLUiR and AmaLgam on the projects used by LOBSTER and LtR and vice versa. LtR's top-N values for Tomcat were computed from the ranking results published in LtR's online annex (Wang and Lo 2014).

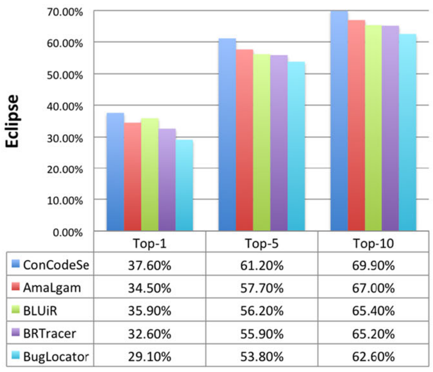
LtR also used AspectJ, Eclipse and SWT but with a different dataset to that of BugLocator. The online annex only included the Tomcat source code, so we were unable to rank the BRs for the other projects with LtR.

Figures 3 and 4 show that except for AmaLgam's Top-1 performance on AspectJ, ConCodeSe outperforms or equals all tools on all metrics for all projects, including BugLocator's MAP for ZXing, which BLUiR and AmaLgam weren't able to match.

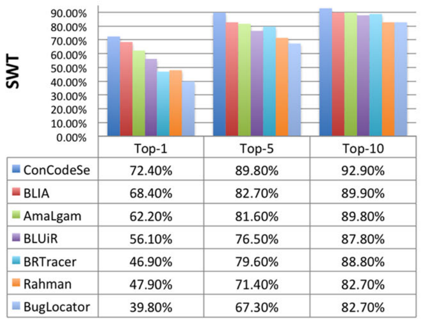
For LOBSTER, the authors report MAP and MRR values obtained by varying the similarity distance in their approach, and we took their best values (0.17 and 0.25). LOBSTER only investigates the added value of stack traces so to compare like for



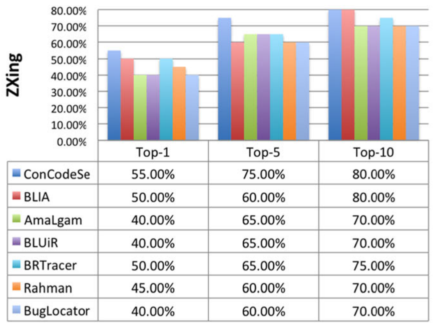
	Top-1	Top-5	Top-10
ConCodeSe	42.30%	68.20%	78.30%
BLIA	37.70%	64.40%	73.20%
AmaLgam	44.40%	65.40%	73.10%
BLUiR	33.90%	52.40%	61.50%
BRTracer	39.50%	60.50%	68.90%
BugLocator	30.80%	51.00%	59.40%



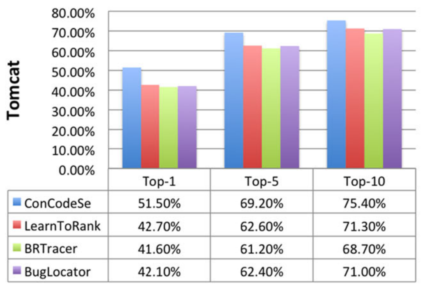
	Top-1	Top-5	Top-10
ConCodeSe	37.60%	61.20%	69.90%
AmaLgam	34.50%	57.70%	67.00%
BLUiR	35.90%	56.20%	65.40%
BRTracer	32.60%	55.90%	65.20%
BugLocator	29.10%	53.80%	62.60%



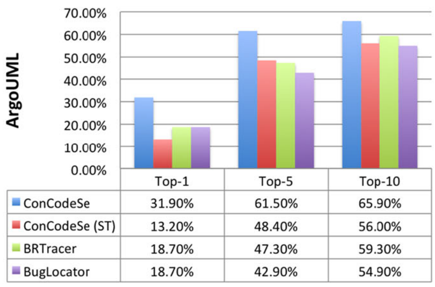
	Top-1	Top-5	Top-10
ConCodeSe	72.40%	89.80%	92.90%
BLIA	68.40%	82.70%	89.90%
AmaLgam	62.20%	81.60%	89.80%
BLUiR	56.10%	76.50%	87.80%
BRTracer	46.90%	79.60%	88.80%
Rahman	47.90%	71.40%	82.70%
BugLocator	39.80%	67.30%	82.70%



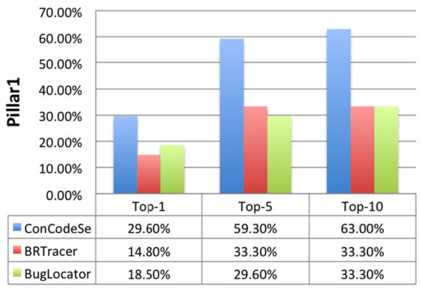
	Top-1	Top-5	Top-10
ConCodeSe	55.00%	75.00%	80.00%
BLIA	50.00%	60.00%	80.00%
AmaLgam	40.00%	65.00%	70.00%
BLUiR	40.00%	65.00%	70.00%
BRTracer	50.00%	65.00%	75.00%
Rahman	45.00%	60.00%	70.00%
BugLocator	40.00%	60.00%	70.00%



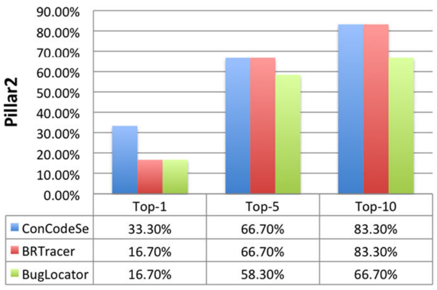
	Top-1	Top-5	Top-10
ConCodeSe	51.50%	69.20%	75.40%
LearnToRank	42.70%	62.60%	71.30%
BRTracer	41.60%	61.20%	68.70%
BugLocator	42.10%	62.40%	71.00%



	Top-1	Top-5	Top-10
ConCodeSe	31.90%	61.50%	65.90%
ConCodeSe (ST)	13.20%	48.40%	56.00%
BRTracer	18.70%	47.30%	59.30%
BugLocator	18.70%	42.90%	54.90%



	Top-1	Top-5	Top-10
ConCodeSe	29.60%	59.30%	63.00%
BRTracer	14.80%	33.30%	33.30%
BugLocator	18.50%	29.60%	33.30%



	Top-1	Top-5	Top-10
ConCodeSe	33.30%	66.70%	83.30%
BRTracer	16.70%	66.70%	83.30%
BugLocator	16.70%	58.30%	66.70%

Fig. 4 Top-N values of the tools

like, we ran ConCodeSe on their ArgoUML dataset using only the ST scoring and still improved on their results (Fig. 3 ConCodeSe-(ST) row).

We note that ConCodeSe always improves the MRR value, which is an indication of how many files a developer has at most to go through in the ranked list before finding one that needs to be changed. User studies (Xia et al. 2016; Kochhar et al. 2016) indicate that developers only look at the top-10, and especially the top-5, results. ConCodeSe has the best top-5 and top-10 results across all projects.

We also get distinctly better results than Rahman et al. (2015), the only other approach to explicitly use file names found in BRs. Looking at the largest project, Eclipse (Fig.4), we note that even a small 1.7% Top-1 improvement over the second best approach (BLUiR) represents 52 more BRs for which the first recommended file is relevant, thus helping developers save time.

We notice that our tool performs almost 2% worse than AmaLgam (42.3 vs 44.4%) for AspectJ when placing relevant files in top-1. Investigating the reasons for this, we found out that in 2 AspectJ BRs, with a FileNotFound exception reported, the changed file got ranked 2nd despite being listed in the ST. This is because the ST lists a utility file in the 1st position and the affected file in the 2nd position. For example, in AspectJ BR #123212, the file AjBuildManager.java uses FileUtil.java to write information to an external file and the FileNotFoundException is thrown by FileUtil first and then propagated to its users like AjBuildManager. Since ST scores are assigned based on the order of each file appearing in the ST, in the case of AspectJ BR #123212, FileUtil gets a higher score than AjBuildManager. To handle this scenario, we experimented by adjusting our ST scoring values but the overall results deteriorated.

In the case of Pillar1, our tool achieves a significantly higher performance over BugLocator and BRTracer in all top-N categories (Fig. 4). It is also interesting to see that BugLocator outperforms BRTracer in the top-1 and top-5 metrics despite that BRTracer is reported to be an improvement over BugLocator. In the case of Pillar2, although our approach achieves identical performance to the second best in the top-5 and top-10 metrics, it is far superior in the top-1 metric and thus outperforms the other approaches in terms of MAP and MRR (Fig. 3).

The first research question is about leveraging filenames. So, we divided each project's BRs in two sets—those BRs mentioning at least one file and those mentioning no files at all—and ran ConCodeSe, BRTracer and BugLocator on each set. The results are in Tables 12 and 13. Table 13 shows that ConCodeSe matches or outperforms the other two tools on every project and metric, except Eclipse's top-1 and SWT's Top-5, Top-10 and MAP. This is evidence that our approach performs well even if a bug report doesn't mention file names.

As a further example, SWT BR #58185 mentions no files and yet ConCodeSe places 3 of the 4 relevant files in the top-5, whereas BugLocator and BRTracer only place 2 files. Similarly, AspectJ BR #95529 contains no file names but out of the 11 relevant files, ConCodeSe ranks 3 files in the top-5 and 1 in the top-10 whereas BugLocator and BRTracer only rank 1 relevant file in the top-5. In all these cases the KP and ST scores are zero and Algorithm 1 uses the lexical (TT) score. The TT and VSM scores are computed with and without using the file's comments, with and without stemming. Thus, even if a bug report doesn't mention file names, our approach still obtains $2*2*2 = 8$ ranks for each file, to choose the best of them.

Table 12 Performance on BRs mentioning at least one file name

Project	#BRs	Tools	Top-1 (%)	Top-5 (%)	Top-10 (%)	MAP	MRR
AspectJ	282	**ConCodeSe**	**42**	**69**	**78**	**0.34**	**0.70**
		BRTracer	33	59	72	0.24	0.54
		BugLocator	18	42	52	0.17	0.34
Eclipse	3043	**ConCodeSe**	**38**	**61**	**70**	**0.37**	**0.57**
		BRTracer	32	57	66	0.33	0.51
		BugLocator	32	55	65	0.32	0.51
SWT	93	**ConCodeSe**	**73**	**89**	**94**	**0.71**	**0.96**
		BRTracer	50	79	88	0.54	0.71
		BugLocator	44	71	84	0.50	0.67
Zxing	19	**ConCodeSe**	**53**	**74**	**79**	**0.54**	**0.65**
		BRTracer	53	68	74	0.51	0.60
		BugLocator	42	68	74	0.44	0.52
Tomcat	942	**ConCodeSe**	**54**	**72**	**79**	**0.56**	**0.71**
		BRTracer	37	59	65	0.40	0.47
		BugLocator	0	64	74	0.28	0.35
ArgoUML	61	**ConCodeSe**	**29**	**57**	**60**	**0.30**	**0.57**
		BRTracer	23	50	65	0.20	0.40
		BugLocator	21	45	57	0.20	0.42
Pillar1	19	**ConCodeSe**	**47**	**58**	**58**	**0.35**	**0.96**
		BRTracer	16	37	37	0.16	0.36
		BugLocator	21	37	42	0.19	0.43
Pillar2	0	**all tools**	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**

The best approach for each project is in bold

Having always 8 ranks to choose from also helps with the other extreme: the bug report includes many file names, but most are irrelevant. For example, SWT BR #83699 mentions 14 files but only 1 is relevant. We rank it in the 4th position using the file's comments, whereas BugLocator and BRTracer rank it 9th and 14th, respectively. Similarly, AspectJ BR #46298 has a very verbose description that mentions 9 files, none relevant. We list the only relevant file in the 2nd position using VSM and comments; BugLocator and BRTracer list it in 6th and 12th position respectively.

More generally, Table 12 shows that ConCodeSe outperforms BugLocator and BRTracer in every metric for every project when the BR contains file names, whether relevant or not. We have separated the BRs that have one or more relevant file names (Table 14) from those that have no relevant file names at all (Table 15). Table 14 shows that if there is at least one relevant file name in the BR, ConCodeSe matches or outperforms the other two tools in every metric for every project. Table 15 shows that the same happens, except for Eclipse, when the BR doesn't include any relevant file name. Even in the case of Eclipse the difference to BugLocator (which outperforms BRTracer) is small. Comparing both tables, one can see that for each project except

Table 13 Performance on BRs mentioning no file name

Project	#BRs	Tools	Top-1 (%)	Top-5 (%)	Top-10 (%)	MAP	MRR
AspectJ	4	**ConCodeSe**	**25**	**75**	**75**	**0.27**	**0.62**
		BRTracer	25	50	50	0.20	0.40
		BugLocator	0	0	25	0.05	0.10
Eclipse	32	**ConCodeSe**	**9**	**34**	**50**	**0.14**	**0.30**
		BRTracer	16	34	50	0.15	0.29
		BugLocator	13	31	47	0.14	0.27
SWT	5	**ConCodeSe**	**40**	**80**	**80**	**0.27**	**0.98**
		BRTracer	0	80	80	0.19	0.62
		BugLocator	40	100	100	0.32	0.90
Zxing	1	**ConCodeSe**	**100**	**100**	**100**	**0.70**	**1.00**
		BRTracer	0	0	100	0.15	0.23
		BugLocator	0	100	100	0.21	0.31
Tomcat	114	**ConCodeSe**	**24**	**39**	**45**	**0.22**	**0.41**
		BRTracer	10	21	32	0.11	0.16
		BugLocator	0	26	40	0.11	0.21
ArgoUML	30	**ConCodeSe**	**38**	**72**	**79**	**0.34**	**0.62**
		BRTracer	10	45	52	0.15	0.33
		BugLocator	14	38	55	0.17	0.36
Pillar1	8	**ConCodeSe**	**13**	**63**	**75**	**0.17**	**0.49**
		BRTracer	13	13	13	0.09	0.18
		BugLocator	13	13	13	0.14	0.23
Pillar2	12	**ConCodeSe**	**33**	**67**	**83**	**0.26**	**0.69**
		BRTracer	17	67	83	0.17	0.61
		BugLocator	17	58	67	0.17	0.61

ConCodeSe's results are in bold

SWT, most BRs do not mention any relevant file and yet ConCodeSe performs best in almost every case.

A BR can be very terse, e.g. a short sentence in the summary and an empty description field, like SWT BR #89533. In this example our tool ranks the only relevant file in 3rd position, by applying VSM and stemming to the BR summary and the file's comments, whereas BugLocator and BRTracer rank the same file in 305th and 19th position respectively. Similarly, for AspectJ BR #39436, the only relevant file is ranked in the top-5, based on the comments in the file, whereas BugLocator and BRTracer rank the same file below the top-10.

Figure 4 only counts for how many BRs at least one affected file was placed in the top-N. The MAP and MRR values indicate that ConCodeSe tends to perform better for each BR compared to other tools, so we also analysed the per-BR performance to measure the number of files per BR placed in the Top-10. This analysis required access to per-BR results and the only publicly available tools are BugLocator and BRTracer.

Table 14 Performance on BRs mentioning at least one relevant file name

Project	#BRs	Tools	Top-1 (%)	Top-5 (%)	Top-10 (%)	MAP	MRR
AspectJ	124	**ConCodeSe**	**58**	**92**	**96**	**0.48**	**0.91**
		BRTracer	44	78	90	0.34	0.68
		BugLocator	30	53	64	0.26	0.46
Eclipse	1083	**ConCodeSe**	**71**	**95**	**98**	**0.62**	**0.93**
		BRTracer	56	84	91	0.53	0.78
		BugLocator	52	77	86	0.49	0.74
SWT	68	**ConCodeSe**	**88**	**99**	**100**	**0.83**	**1.00**
		BRTracer	59	91	96	0.63	0.80
		BugLocator	57	79	91	0.59	0.78
Zxing	9	**ConCodeSe**	**89**	**100**	**100**	**0.86**	**0.97**
		BRTracer	78	89	89	0.75	0.84
		BugLocator	67	89	89	0.66	0.75
Tomcat	524	**ConCodeSe**	**81**	**92**	**93**	**0.78**	**0.95**
		BRTracer	56	77	81	0.57	0.65
		BugLocator	0	86	91	0.38	0.46
ArgoUML	12	**ConCodeSe**	**58**	**100**	**100**	**0.56**	**0.86**
		BRTracer	50	100	100	0.46	0.68
		BugLocator	33	75	83	0.40	0.54
Pillar1	4	**ConCodeSe**	**100**	**100**	**100**	**0.59**	**1.00**
		BRTracer	50	100	100	0.56	0.82
		BugLocator	75	100	100	0.59	1.00
Pillar2	0	**all tools**	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**

The best approach for each project is in bold

Table 16 shows, for example, that for 128/286 = 45% (resp. 85/286 = 30%) of AspectJ's BRs, our tool placed more files in the top-10 than BugLocator (resp. BRTracer). This includes BRs in which ConCodeSe placed at least one and the other tools placed none. The 'same' columns indicate the percentage of BRs for which both tools placed the same number of affected files in the top-10. This includes cases where all approaches can be improved (because neither ranks an affected file in the top-10), and where none can be improved (because all tools place all the affected files in the top-10).

From all the results shown we can answer RQ1 affirmatively: leveraging the occurrence of file names in BRs leads in most cases to better performance than using project history. Due to the use of the best of 8 scores, ConCodeSe is robust to the absence of file names or the presence of irrelevant file names in the BRs.

4.2 RQ2: scoring without similar bugs

Our second research question was to *evaluate the contribution of using similar bug reports.* As described earlier, BugLocator, BRTracer, BLUiR and AmaLgam utilise a

Table 15 Performance on BRs mentioning no relevant file name

Project	#BRs	Tools	Top-1 (%)	Top-5 (%)	Top-10 (%)	MAP	MRR
AspectJ	162	**ConCodeSe**	**30**	**52**	**65**	**0.24**	**0.54**
		BRTracer	25	43	57	0.17	0.42
		BugLocator	9	33	42	0.11	0.25
Eclipse	1992	**ConCodeSe**	**19**	**43**	**54**	**0.22**	**0.37**
		BRTracer	20	41	52	0.22	0.36
		BugLocator	21	43	53	0.23	0.38
SWT	30	**ConCodeSe**	**33**	**67**	**77**	**0.36**	**0.66**
		BRTracer	20	50	70	0.30	0.49
		BugLocator	13	57	70	0.26	0.46
Zxing	11	**ConCodeSe**	**27**	**55**	**64**	**0.29**	**0.43**
		BRTracer	27	46	64	0.29	0.37
		BugLocator	18	55	64	0.23	0.32
Tomcat	532	**ConCodeSe**	**22**	**46**	**58**	**0.27**	**0.40**
		BRTracer	13	33	42	0.18	0.22
		BugLocator	0	33	49	0.14	0.20
ArgoUML	79	**ConCodeSe**	**28**	**56**	**61**	**0.27**	**0.55**
		BRTracer	14	41	54	0.15	0.34
		BugLocator	17	38	52	0.16	0.38
Pillar1	23	**ConCodeSe**	**26**	**52**	**57**	**0.22**	**0.69**
		BRTracer	9	17	17	0.06	0.22
		BugLocator	9	17	17	0.10	0.26
Pillar2	12	**ConCodeSe**	**33**	**67**	**83**	**0.26**	**0.69**
		BRTracer	17	67	83	0.17	0.61
		BugLocator	17	58	67	0.17	0.61

ConCodeSe's results are in bold

Table 16 ConCodeSe versus BugLocator and BRTracer per-query top-10 performance

Project	BugLocator			BRTracer		
	Better (%)	Same (%)	Worse (%)	Better (%)	Same (%)	Worse (%)
AspectJ	45	48	7	30	55	15
Eclipse	23	64	13	22	65	13
SWT	24	69	6	21	74	4
ZXing	10	85	5	20	75	5
Tomcat	22	66	12	56	37	7
ArgoUML	25	62	13	29	59	12
Pillar1	48	48	4	48	48	4
Pillar2	58	33	8	50	33	17

Table 17 Similar bug reports examples (from SWT)

BR#	Affected Java Files	BugLocator		BRTracer		BLUiR		ConCodeSe
		rVSM	SimiScore	no SimiScore	SimiScore	structure	SimiScore	
78856	OS.java	37	36	89	88	3	6	1
79419	Link.java	18	18	4	4	31	6	1
	OS.java	58	58	48	47	11	21	1
83262	RTFTransfer	224	224	214	214	43	79	35
	TextTransfer	202	202	198	197	–	–	36
87676	Tree.java	49	21	10	6	4	3	2

feature called SimiScore, which uses the BR terms to find similar closed BRs. The files changed to fix those BRs are suggested as likely candidates for the current BR being searched. To answer RQ2 we ran BugLocator and BRTracer with and without SimiScore, as explained at the start of Sect. 4.

Unfortunately, we were unable to obtain BLUiR and AmaLgam to perform runs without SimiScore, but we do not consider this to be a handicap because from the published results it seems that SimiScore benefits mostly BugLocator and BRTracer. We selected the SWT BRs reported in the BLUiR paper (#78856, #79419, #83262 and #87676) and then ran BugLocator and BRTracer on the same BRs to compare their performance.

As shown in Table 17, BugLocator placed the file Tree.java in the 49th and 21st positions in the ranked list by using their revised VSM (rVSM) approach first and then by considering similar BRs. In the case of BRTracer, the introduced segmentation approach already ranked the file in the top-10 (10th position) and SimiScore placed the same file at an even higher position (6th). In the case of BLUiR, the same file is placed at 4th and 3rd positions, respectively. For the other cases in the table, SimiScore doesn't improve (or only slightly so) the scoring for BugLocator. In the case of BLUiR, apart from the great improvement for Link.java, SimiScore leads to a lower rank than structural IR.

We ran BugLocator and BRTracer without using SimiScore on all projects, to have a more like-for-like comparison with ConCodeSe in terms of input (no past BRs). Comparing Fig. 4 (with SimiScore) and Table 18 (without) shows a noticeable performance decline in BugLocator and BRTracer when not using similar BRs and thus an even greater improvement achieved by ConCodeSe. BLUiR without SimiScore also outperforms BugLocator and BRTracer with SimiScore. Interestingly BRTracer and ConCodeSe perform equally well in top-5 and top-10 for Pillar2.

We answer RQ2 by saying that although the contribution of similar BRs significantly improves the performance of BugLocator and BRTracer, it is not enough to outperform ConCodeSe. The large contribution of SimiScore for BugLocator and BRTracer is mainly due to the lower baseline provided by rVSM, as reinforced by the results in the next section.

Table 18 BugLocator and BRTracer without similar bug reports score

Project	Approach	Top-1 (%)	Top-5 (%)	Top-10 (%)	MAP	MRR
AspectJ	BugLocator	22.7	40.9	55.6	0.19	0.18
	BRTracer	38.8	58.7	66.8	0.27	0.47
	ConCodeSe	**42.3**	**68.2**	**78.3**	**0.33**	**0.67**
Eclipse	BugLocator	24.4	46.1	55.9	0.26	0.35
	BRTracer	29.6	51.9	61.8	0.30	0.40
	ConCodeSe	**37.6**	**61.2**	**69.9**	**0.37**	**0.57**
SWT	BugLocator	31.6	65.3	77.6	0.40	0.47
	BRTracer	46.9	79.6	88.8	0.53	0.59
	ConCodeSe	**72.4**	**89.8**	**92.9**	**0.68**	**0.94**
ZXing	BugLocator	40.0	55.0	70.0	0.41	0.48
	BRTracer	45.0	70.0	75.0	0.46	0.55
	ConCodeSe	**55.0**	**75.0**	**80.0**	**0.55**	**0.68**
Tomcat	BugLocator	42.1	62.4	71.0	0.26	0.33
	BRTracer	36.6	57.3	65.6	0.45	0.56
	ConCodeSe	**49.9**	**69.2**	**75.4**	**0.52**	**0.66**
ArgoUML	BugLocator	18.7	42.9	54.9	0.11	0.48
	BRTracer	18.7	46.2	54.9	0.20	0.38
	ConCodeSe	**31.9**	**61.5**	**65.9**	**0.30**	**0.55**
Pillar1	BugLocator	18.5	29.6	33.3	0.17	0.37
	BRTracer	14.8	29.6	29.6	0.14	0.31
	ConCodeSe	**29.6**	**59.3**	**63.0**	**0.22**	**0.69**
Pillar2	BugLocator	16.7	58.3	66.7	0.17	0.61
	BRTracer	16.7	66.7	83.3	0.17	0.61
	ConCodeSe	**33.3**	**66.7**	**83.3**	**0.26**	**0.69**
Average	BugLocator	27	50	61	0.25	0.40
	BRTracer	31	57	65	0.33	0.50
	ConCodeSe	**44**	**69**	**76**	**0.40**	**0.57**

ConCodeSe's results are in bold

4.3 RQ3: VSM's contribution

As described in Sect. 3.2, 4 of a file's 8 rankings are obtained with the VSM probabilistic method available in the Lucene library, and the other 4 with our lexical similarity ranking method (Algorithm 1). To find out the added value of VSM, we performed two runs, one with VSM only and the other with lexical similarity scoring only (Table 19).

VSM outperforms the lexical scoring for the larger projects (AspectJ and Eclipse), i.e. with most code files (Table 1), and underperforms for small projects (SWT, ZXing and Pillar2). In cases of medium size projects (Tomcat and ArgoUML), lexical scoring outperforms VSM for Tomcat and underperforms for ArgoUML. As to be expected, each scoring by itself has a poorer performance than ConCodeSe, which combines both, whilst the total number of BRs located by ConCodeSe (Fig. 4) is not the sum of

Table 19 Lucene VSM versus lexical similarity scoring

Project	Approach	Top-1 (%)	Top-5 (%)	Top-10 (%)	MAP	MRR
AspectJ	**VSM**	**28.0**	**47.2**	**60.1**	**0.22**	**0.49**
	Lexical similarity	20.6	44.4	59.4	0.20	0.39
Eclipse	**VSM**	**23.3**	**45.7**	**56.6**	**0.25**	**0.41**
	Lexical similarity	18.6	36.3	48.2	0.21	0.32
SWT	VSM	39.8	71.4	82.7	0.46	0.66
	Lexical similarity	**63.3**	**79.6**	**89.8**	**0.60**	**0.82**
ZXing	VSM	35.0	55.0	65.0	0.37	0.45
	Lexical similarity	**40.0**	**70.0**	**80.0**	**0.47**	**0.54**
Tomcat	VSM	28.1	47.7	57.6	0.32	0.43
	Lexical similarity	**34.0**	**56.6**	**64.0**	**0.38**	**0.48**
ArgoUML	**VSM**	**24.2**	**51.6**	**58.2**	**0.24**	**0.45**
	Lexical similarity	13.2	48.4	56.0	0.20	0.33
Pillar1	**VSM**	**25.9**	**48.1**	**55.6**	**0.18**	**0.59**
	Lexical similarity	7.4	33.3	40.7	0.10	0.36
Pillar2	VSM	8.3	66.7	75.0	0.18	0.62
	Lexical similarity	**25.0**	**66.7**	**75.0**	**0.20**	**0.71**

The best approach for each project is in bold

the parts. In other words, many files are ranked in the top-N by both scoring methods, and each is able to locate BRs the other can't. ConCodeSe literally takes the best of both worlds.

For all projects except ArgoUML, VSM by itself performs poorer than the best other approach (Table 4), which shows the crucial contribution of lexical similarity scoring for the improved performance of our approach.

The VSM variant we adopt outperforms in many cases rVSM, introduced in BugLocator and also utilised in BRTracer. Even ConCodeSe's lexical similarity by itself outperforms rVSM in most cases. This can be seen by comparing the VSM (or lexical similarity) rows of Table 19 against the BugLocator rows of Table 18, where SimiScore is turned off to only use rVSM.

We thus answer RQ3 by concluding that VSM provides a bigger contribution for projects with a large number of files, which makes the use of term and document frequency more meaningful. We also confirm that the exact IR variant used is paramount (Saha et al. 2013): Lucene's VSM and our simple lexical matching outperform BugLocator's bespoke rVSM in many cases as well as BLUiR's Okapi, especially for SWT and ZXing. However, VSM on its own isn't enough to outperform other approaches.

5 RQ4: user study

In the previous section, we evaluated our approach with a range of open source projects and showed it outperformed current state-of-the-art tools in a simpler, faster, and more general way that doesn't require history. To investigate the generalisability of our approach, we conducted user studies in three different companies with professional developers. The first aim of the study was to demonstrate the applicability of our approach in industrial environments. It may be that commercial applications and bug reports may differ from those used in the previous section, thus impacting on the performance of our approach.

Previous user studies, summarised in Sect. 2.4, reveal that: large search results returned by the integrated development environment (IDE) tools cause developers to analyse several files before performing bug-fixing tasks and that providing developers with automated code search tools that present results in a ranked order would be of great benefit in performing their daily tasks (Starke et al. 2009; Sillito et al. 2008); 5 is the magic number of results that developers deem acceptable to inspect (Xia et al. 2016; Kochhar et al. 2016); almost all developers 'refuse' to inspect more than 10 items (Kochhar et al. 2016); items are not inspected in the order they are ranked (Parnin and Orso 2011). None of those studies used IR-based bug localisation. Thus the second aim of our study was to see how developers perceive and use the ranked search results of ConCodeSe.

Hence we ask our fourth RQ as:

How does our approach perform with industrial applications and does it benefit developers by presenting the search results in ranked order of relevance?

In order to not require users to modify ConCodeSe to interface with their software repositories and issue trackers, we implemented a simple front-end (GUI) panel to our tool for users to paste the summary and description of a BR and search for the candidate source code files. The results are displayed in the ranked order of relevance from most to least likely.

5.1 Study design

Based on the work of Starke et al. (2009) and Sillito et al. (2008), we designed a study with professional software developers with their a priori consent on the following:

1. Data collected:
   (a) How useful was the ranked list?
   (b) How accurate was the ranking?
   (c) How much were you confident on the list?
   (d) How intuitive was it compared to an IDE search tool?
   (e) What did work best?
2. Data collection method:
   (a) Pre-session interview
   (b) Each search session: choosing the BRs, running the tool, evaluating the results.
   (c) Post-session interview

The first author contacted five different companies where he formerly worked as a freelance software developer, explaining the study. He sent each company an information leaflet explaining that we were looking for participants to take part in a study where our tool will be used to search source code files of a software application with descriptions available in a BR document. The leaflet informed interested parties how the study would unfold:

Table 20 Participant profile

Company	Business nature	Study dates (2015)	Participant Id	Years of experience	Years of IDE Experience	Time spent on maintenance (%)
U	Finance	29.06–03.07	1	20	10	30
			2	15	10	70
S	Logistic	06.07–04.08	3	9	9	20
A	Software Services	05.06–16.10	4	11	11	40

1. We would conduct a 30–45 min pre-session interview to explain how to use ConCodeSe.
2. Afterwards, the participants would be required to try the tool for 7–10 business days in their own time and document their experience on the relevance of the suggested source files for the search tasks performed.
3. Finally, at the end of the trial period a post-session interview lasting for 30–45 min would be conducted to collect details on their usage experience.

The leaflet further explained that the participation would be treated in strict confidence in accordance with the UK Data Protection Act and no personal information could be passed to anyone outside the research team. It also informed that we would aim to publish the findings from the study, but no individual would be identifiable. Participants were allowed to answer questions with "not applicable" if they did not intend to provide an answer during both interview sessions.

Out of the five contacted companies, only three agreed to participate and two of them agreed to have the post-session interview session recorded as video. Upon receiving the participation agreement from each company, the first author obtained written consent from each participant prior to the study. Table 20 shows the participant profile information at each company. Table 21 details the artefacts used in the study: the size of the industrial applications is comparable to the medium-sized OSS projects (Table 1). All participants were professional software developers and did not require compensation since they were recruited on a volunteering basis.

Although the first author has worked for the three companies in the past, he did not have prior knowledge of the applications used in the study nor had he previous contact with the participants. He went to each company and presented ConCodeSe and the aim of the study, which was to identify the benefits provided by a ranked list of candidate source code files that may be relevant for a BR at hand during software maintenance. Subsequently, he conducted a pre-session interview with the developers

Table 21 Artefacts used

Company	Application in production since	# of source code class files	# of bug reports used
U	2014	2840	10
S	2009	2240	10
A	2013	4560	10

to collect information about their experience and their thought process on how they performed their tasks during daily work. One of the intentions of this pre-session interview was to make developers aware of their working habits so that they could document their experience with our tool more accurately.

5.2 Results

We present the information collected during the pre- and post-interviews. Throughout both sessions developers referred to a source code file as *class* and to a bug as *defect*.

5.2.1 Pre-session interview findings

Based on the observations made by Starke et al. (2009) we designed the pre-interview questions in order to obtain a summative profile of developers and their working context. Below we list the questions asked and the general answers given to each of these questions.

1. *How do you go on about solving defects?* All developers indicated that they read BR descriptions and log files looking for what to fix based on experience. They also try to reproduce the bug to debug the execution steps and look at the source code to see what is wrong. Only one developer indicated that he compares the version of the application where the bug has occurred against a version where the bug hasn't been reported yet to detect any changes that might have caused the reported bug.
2. *How would you go on about solving a defect in an unfamiliar application?* In general developers responded that they search code by using some words (e.g. nouns and action words) from the bug description to pick a starting point. They also read the user guide to understand the behaviour of the application and technical guide to understand the architecture. One developer indicated that he would attempt to simulate the reported behaviour if he could understand the scenario described in the BR and some test data is provided.
3. *What about when you cannot reproduce or haven't got a running system?* At least to get an idea of a starting point, developers look for some hints in class and method names. One developer stated that he would include logging statements in certain classes to print out trace information and collect details of execution in production so that he could see what was happening during run time.
4. *In order to find starting points to investigate further, what kind of search tools do you use?* Although all developers said that they use search functions available in IDEs, e.g. full text, references, inheritance chain and call hierarchies, one said that he prefers to use the Mac OS Spotlight search because in addition to source files, it indexes other available artefacts like the configuration files, GUI files (e.g. html, jsp and jsf) as well as the documentation of the application.
5. *How do you evaluate a call hierarchy?* Developers explained that they would start by performing a '*reference*' search of a class and browse through the results. One said that "*I also look to see if method and variable names that are surrounding the search words also make sense for the bug description that I am involved with*" .

6. *What do you consider as being important, callers of a class or the called classes of a class?* Each reply started with "*That depends on...*". It seems that each participant has a different way of assessing the importance of the call hierarchy. For some developers the importance is based on the bug description, e.g. if the bug description indicated that some back-end modules are the culprit, then they would look to see where the control flows are coming from, while for others, it is based on architecture, e.g. if a certain class is calling many others then they would consider this to be a bad architecture and ignore the caller.
7. *How do you decide which classes to investigate further, i.e. open to look inside, and which to skip?* Almost all developers answered first by saying "*gut feeling*" and then went on to describe that they quickly skim through the results list and look for clues on package or file names to determine whether it makes sense to further investigate the file contents or not.
8. *Do you consider descriptive clues?* Once again developers replied that they rely on their experience of the project to relate conceptual words found in the bug reports to file names.
9. *When do you decide to stop going through the path and start from the beginning?* All participants indicated that after browsing through the search results looking at file names, they may open 3 or 4 files to skim through their content and when no clues were detected, they would decide to start a new search with different words.
10. *What kind of heuristics do you use when considering concepts implemented by neighbouring classes?* In general, developers indicated that they look at the context of a source file within the bug that they are working on, i.e. relevance of a file based on the BR vocabulary. One developer said that project vocabulary sometimes causes ambiguity because in the application he works with, certain file names contain the word *Exception*, which refers to a business exception rule and not to an application exception, i.e. error condition.

The pre-session interview answers confirmed the challenges highlighted by previous studies (Starke et al. 2009; Sillito et al. 2008). We asked developers to try out ConCodeSe by downloading, installing and performing search tasks in their own time. We decided to perform an uncontrolled study because we felt that this would provide a more realistic environment and also allow developers to have adequate time to utilise our tool without negatively impacting their daily workload.

We asked them to collect screen shots showing the information they entered as search text and the results listed. We instructed them to use closed bug reports for search query where the affected files were also documented so that they can prove to us whether the results contained the relevant files or not.

We arranged to meet them in 7–10 business days for a 30–45 min post-session interview to gather their experiences and to inspect the screen shots.

5.2.2 Post-session findings

Below we list the questions asked during the post-interview sessions and the general answers given to each of these questions.

1. *How did you go on using the tool?* All indicated that they first used a few BRs with which they previously worked with to become familiar with our tool and its performance. Afterwards they have randomly chosen closed BRs that they had not previously worked with, performed search by using the words found in the summary and description fields of the BRs and then compared the search results against the files identified as affected in the issue-tracking tool. One developer said, "*I performed search tasks by selecting a few words, i.e. 2–3, from the BR, which I knew would lead to relevant results. However this did not work very well so I have gradually increased the search terms with additional ones found in the BR descriptions. This provided more satisfactory results and the relevant classes started appearing in the result list*".
2. *Did the tool suggest relevant classes in the ranked list?* Inspecting the screen-shots of the results, we found that in Company-U, for 8 out of 10 BRs at least one file was in the top-10. In Company-S, for 9 out of 10 BRs the tool ranked at least one affected file at top-1. Developer in Company-A said that "*The relevant file was always ranked among the top-3. I never needed to investigate files which were ranked beyond top-3*".
3. *Were there relevant classes in the result list, which you might not have thought of without the tool?* All participants replied with '*yes*'. One developer said that "*I have also opened up the files listed at top-2 and top-3, despite that they were not changed as part of the BR at hand. However looking at those files would have also led me to the relevant one, assuming that the relevant file was not in the result list*" . Another indicated that "*a BR description had a file name which got ranked at top-1 but did not get changed. However it was important to consider that class during bug fixing*".
4. *Did the ranked list provide clues to formulate search descriptions differently?* Developers indicated that most of the time they had a clue to what they were looking for but were not always certain. However, seeing file names with a rank allowed them to consider those files with a degree of importance as reflected by the ranking. In case of one defect, the description caused a lot of noise and resulted in the top 5 files being irrelevant, but 3 relevant files were placed in the top-10.
5. *What did you add to or remove from the description to enhance the search?* Developers indicated that the search led to better results when the query included possible file names, exceptions and stack trace information. However, in one case ignoring the description and searching only with the summary resulted in 4 out of 8 relevant files to be ranked at top-5 and 1 file between top-5 and top-10. One developer indicated that in one case despite changing search words, the relevant file was still not found because the BR did not provide any clues at all.
6. *What would you consider to reduce false positives in search results?* In general, all participants suggested that project specific vocabulary mapping should be used to cover cases when concepts in file names cause misleading results. For example, Batch jobs are named as Controller, so when a BR describes a batch job without using the word *controller*, then those classes are not found if there are no comments indicating additional clues. In case of such a defect, only 5 out of 18 affected files were found.

7. *How comfortable were you with the ranked list?* All developers indicated that the tool was easy to use and after performing 3–4 searches to become familiar with the tool, they were satisfied with the results since the files at the first 5 positions were most often relevant ones. Despite their comfort with the tool, all participants indicated that if a BR description contained fewer technical details, e.g. fewer file names, and more descriptive information, e.g. test scenario steps, the tool was not useful. In fact they felt that in such cases any tool would fail to suggest the relevant file(s).
8. *Were you able to get to the other relevant classes based on the suggested ones that were relevant?* All participants expressed that the ranked list helped them to consider other focal points and gave them a feel for what else to look for that might be relevant. One said that "*Most of the cases I browsed through the results and opened only the top-1 and top-2 files to see if they were the relevant ones or to see if they can lead me to the file that may be more appropriate*".
9. *Would you consider such a tool to support you in the current challenges you have?* In general, all indicated that the ranked list would benefit anyone who is new to a project since it could guide novice developer to the parts of the application when searching for files that may be relevant to solve a defect. This would in turn allow the novice developers to rapidly build up application knowledge. One said "*The tool would definitely speed up the learning curve for a new team member who is not familiar with the architecture and code structure of our project. It would save him a lot of time during the first few weeks. After that due to the small size of the project (12000 LOC) it would not provide much significance because the developer would become familiar with the code anyway*". We were told that as a research product our prototype tool was very stable. They experienced no failures, i.e. crashes or error exceptions. One developer said that even when running the tool in a virtual machine (VM) environment, suspending and then resuming the VM, our tool continued to function.

To our final 10th question, "*What would you suggest and recommend for future improvements?*", we have received a lot of valuable suggestions. First of all, we were told by all developers that the biggest help would be to write better bug descriptions and to introduce a defect template to solicitate this. They noticed that defect descriptions containing test steps entered by 1st level support is noisy. For example, in case of one defect, the relevant file is ranked in top-5 and its test file in top-1 so the relevant files were obscured by test files.

Also we were suggested to include in the search the content of configuration files, e.g. config.xml, DB scripts and GUI files. One developer noticed that the words on the first search field (BR summary) are given more importance. He wished that the second field (BR description) is treated with the same importance as the first search field.

In addition we have received several cosmetic suggestions for presenting the results and interacting in search fields, like proposing keywords, e.g. auto complete based on terms found in the source code. Also participants felt that it would be helpful to display, next to the ranked files, the words in those files that matched the BR so that the user can determine whether it really makes sense to investigate the content of the file or not.

Interestingly one developer said that sometimes he did not consider the ranking as an important factor and suggested to group files into packages, based on words matching the package name and then rank the files within that group.

Most of the suggestions concern the developer interface and are outside the scope of ConCodeSe. We intend to investigate whether configuration files can be matched to BRs in future work.

5.3 Evaluation of the results

Modern IDE search tools offer limited lexical similarity function during a search. The developers are required to specify search words precisely in order to obtain accurate results, which may require developers to be familiar with the terminology of the application and the domain. The success of modern IDE search tools depend on the clarity of the search terms otherwise the results may contain many false positives. To compensate for these weaknesses, developers choose to specify on average 3 words (see post-session interview answer 1) when searching for relevant files in IDEs instead of using all the words available in a BR. Since current IDE search tools deprive developers from the advantage of utilising the full information available in BRs, developers may search outside of the IDE (see pre-session interview answer 4).

Furthermore, in current IDEs, the search results are not displayed in a ranked order of relevance causing developers to go through several irrelevant files before finding relevant ones as entry points for performing bug-fixing tasks. Since developers are faced with the challenge of manually analysing a possibly long list of files, they usually tend to quickly browse through the results and decide on its accuracy based on their gut feeling as revealed during our pre-session interview. They also prefer to perform a new search query using different words rather than opening some files to investigate their content. These repetitive search tasks cost additional effort and add burden on the productivity of the developers causing them to lose focus and introduce errors due to fatigue or lack of application knowledge.

We set out to explore whether our ranking approach would benefit developers. Based on the post-session interview answers provided by developers working in different industrial environments with different applications, we confirm that developers welcomed the ranked result list and stated that since most of the relevant files were positioned in the top-5, they were able to avoid the error prone tasks of browsing long result lists and repetitive search queries by focusing on the top-5 portion of the search results.

We were interested in finding out whether the ranked list would point developers to other files that might be of importance but were not initially thought to be relevant. After trying out our tool, at the post-session interview, the developers said that the result list contained other relevant files that they would not have thought of on their own without our tool (see post-session interview answer 3). Developers stated that those additional and not thought of files would not have appeared in the result of the IDE search tool they use because those files do not contain the search terms. The use of stemming allows us to still match the bug report to the code file for some of those cases.

Finally we wanted to see whether our tool, which leverages the textual information available in bug reports, encourages developers to use the full description of a BR when formulating search queries. During the pre-session interview, developers told us that they use their gut feeling and experience when selecting words to use on the search query. At the post-session interview, we were told by developers that incrementing the search words with additional ones from the BRs improved the results.

In software projects, a developer may get assistance from other team members or expert users when selecting the initial entry points to perform the assigned maintenance tasks. We were told that our tool complements this by providing a more sophisticated search during software maintenance. Thus the BR vocabulary can be seen as the assistance provided by the expert team members and the file names in the search results can be seen as the initial entry points to investigate additional relevant files.

Furthermore, we were told that search results still depend on the quality of bug descriptions. In case of tersely described BRs, even experienced developers find it challenging to search for relevant files. In addition, we found that BRs can be of (1) developer nature with technical details, i.e. references to code files and stack traces, or (2) descriptive nature with business terminology, i.e. use of test case scenarios. Since BR documents may come from a group of people who are unfamiliar with the vocabulary used in the source code, we propose that BR descriptions contain a section for describing the relevant domain vocabulary. For example a list of domain terms implemented by an application can be semi-automatically extracted and imported into the BR management tool. Subsequently, when creating a BR, the user may choose from the list of relevant domain terms or the tool may intelligently suggest the terms for selection.

From all these results, we can answer RQ4 affirmatively: in case of business applications, our tool also achieves to place at least one file into top-10 for on average 88% of the BRs (see post-session interview answer 2 and Fig. 4: 83% in Pillar2, 80% in Company U, 90% in S and 100% in A). Our study also confirms that users will focus on the first 10 suggestions and that therefore presenting the search results ranked in the order of relevance for the task at hand benefits developers.

6 Discussion

It has been argued that textual information in BR documents is noisy (Zimmermann et al. 2010). Our approach is partly based on focussing on one single type of information in the BR: the occurrence of file names. The assumption is that if a file is mentioned in the BR, it likely needs to be changed to fix the bug. Manual inspection of only 9.4% of the BRs reinforced that assumption and revealed particular positions in the summary and in the stack trace where an affected file occurs. Our improved results provide further evidence of the relevance of the assumption.

On the other hand, it has been argued that if bug reports already mention the relevant files, then automated bug localisation provides little benefit (Kochhar et al. 2014). This would indeed be the case if almost all files mentioned in a bug report were true positives. As Table 3 shows, most bug reports only affect a very small number of files, and yet they may mention many more files, especially if they contain stack traces

(Table 6). In Sect. 4.1.4 we gave two examples of such BRs, both placed in the top-5 by ConCodeSe: one mentioning 9 files, all irrelevant, the other mentioning 14, of which only one was relevant. Developers are interested in finding out focus points and how files relate to each other, especially when working in unfamiliar applications (Sillito et al. 2008). Automated bug localisation can help developers separate the wheat from the chaff when looking at the files mentioned in bug reports, and even suggest relevant files not mentioned in the reports.

We offer some reasons why we think our approach works better. First, other approaches combine scores using weight factors that are fixed. We instead take always the best of several ranks for each file. In this way, we are not a priori fixing for each file whether the lexical scoring or the probabilistic VSM score should take precedence. We also make sure that stemming and comments are only taken into account for files where it matters. The use of the best of 8 scores is likely the reason for improving the key MRR metric across all projects.

Second, we leverage structure further than other approaches. Like BLUiR, we distinguish the BR's summary and description, but whereas BLUiR treats each BR field in exactly the same way (both are scored by Indri against the parts of a file) we treat each field differently, through the key position and stack trace scoring.

Third, our approach simulates the selective way developers scan BRs better than other approaches. It has been argued that developers may not necessarily need automated bug localisation tools when file names are present in the bug reports (Kochhar et al. 2014) because they may be able to select which information in the bug report is more relevant, based on the structure of the sentences and their understanding of the reported bug (Wang et al. 2015). ConCodeSe first looks at certain positions of the bug report and then, if unsuccessful, uses all terms in the BR, stopping the scoring when a full file name match is found. However, the same cannot be said for most automated tools: they always score a file against all the BR terms, which may deteriorate the performance if a file has more matching terms with the bug report, as it is scored higher and falsely ranked as more relevant (Moreno et al. 2014).

Fourth, our approach addresses both the developer nature and the descriptive nature of BRs, which we observed in the analysis of the BRs for these projects and in particular for Pillar1 and Pillar2 (Dilshener and Wermelinger 2011). BRs of a developer nature tend to include technical details, like stack traces and class or method names, whereas BRs of a descriptive nature tend to contain user domain vocabulary. By leveraging file (class) names and stack traces when they occur in BRs, and by otherwise falling back to VSM and a base lexical similarity scoring (Algorithm 2), we cater for both types of BRs. As the non-bold rows of Table 10 show, the fall-back scoring alone outperforms all other approaches in terms of MAP and MRR (except AmaLgam's MAP score for AspectJ).

Although LOBSTER doesn't use historical information either, it was a study on the value of stack traces over VSM, and thus only processes a subset of developer-type BRs, those with stack traces.

To sum up, we treat each BR and file individually, using the summary, stack trace, stemming, comments and file names only when available and relevant, i.e. when they improve the ranking. This enables our approach to deal with very terse BRs and with BRs that don't mention files.

As for the efficiency of our approach, creating the corpus from the application's source code and BRs takes, on a machine with a 3GHz i3 dual-core processor and 4GB RAM, 3 h for Eclipse (the largest project, see Table 1). Ranking (8 times!) its 12863 files for all 3075 BRs takes about 1.5 h, i.e. on average almost 2 seconds per BR. We consider this to be acceptable since our tool is a proof of concept and developers locate one BR at a time.

6.1 Threats to validity

The construct validity addresses how well an experiment set up measures to its claims. During creation of the searchable corpus, we relied on the JIM tool to extract the terms from source code identifiers and on regular expressions for extracting the stack trace from the BR descriptions. It is possible that other tools may produce different results. Also, the queries (i.e. search terms) in our study were taken directly from BRs. Our user study showed that the developers formulate their queries differently when locating bugs in an IDE and that the use of different queries with vocabularies more in line with the source code would yield better results. However, using the BR summaries and descriptions as queries instead of manually formulated queries avoided the introduction of bias from the authors.

In the user study, the bug localisation was uncontrolled, to avoid disturbing the daily activities of the developers. This may be a potential threat to construct validity. We partially catered for this threat by asking developers to make screen shots of the results, which they showed during the post-session interview. In future we aim to conduct an ethnographic study to observe how developers use the tools for live bug localisation tasks using our tool against using no tool or other tools. The study would be both qualitative, e.g. whether developers' bug localization strategies depend on the tool used or its absence, and quantitative, e.g. to compare the time taken to fix bugs.

The internal validity addresses the relationship between the cause and the effect of the results to verify that the observed outcomes are the natural product of the implementation. We catered for this by comparing the search performance of ConCodeSe like for like (i.e., using the same datasets and the same criteria) with eight existing bug localisation approaches (Zhou et al. 2012; Saha et al. 2013; Moreno et al. 2014; Wong et al. 2014; Wang and Lo 2014; Ye et al. 2014; Youm et al. 2015; Rahman et al. 2015) as well as assessing the contribution of the off-the-shelf Lucene library's VSM. Therefore, the improvement in results can only be due to our approach. It is conceivable that an IR engine using the LSI model may produce more or less sensitive results to using file names in BRs. We plan to experiment with LSI in future work. Also, we used fixed values to score the files, obtained by manually tuning the scoring on AspectJ and SWT. We confirmed the rationale behind those values (namely, distinguish certain positions and assign much higher scores than base term matching) led to the best results, by trying out scoring variations.

The conclusion validity refers to the relationship between the treatment and the outcome and if it is statistically significant. We used a non-parametric Wilcoxon matched pairs statistical test since no assumptions were made about the distribution of the results. This test is quite robust and has been extensively used in the past to con-

Table 22 Wilcoxon test comparison

Statistics	BugLocator		BRTracer	
	Top-1	Top-10	Top-1	Top-10
Z-value	−4.2686	−3.2351	−2.73934	−3.9656
W-value	254	576	487	543
p-value	0.047504	0.0326	0.043432	0.0067

duct similar analyses (Schröter et al. 2010; Moreno et al. 2014). Based on the values obtained as shown in Table 22, we conclude that on average ConCodeSe locates significantly ($p \leq 0.05$) more relevant source files in the top-N, which confirms that the improvement in locating relevant files for a BR in the top-N position by our tool over the state of the art is significant.

The external validity addresses the possibility of applying the study and results to other circumstances. The small size of the user study (4 participants from 3 companies) and the characteristics of the projects (e.g. the domain, the identifier naming conventions, and the way comments and BRs are written, including the positions where file names occur) are a threat to external validity. We reduced this threat by repeating the search experiments with 11 different OSS and industrial applications, developed independently of each other, except for SWT and Eclipse. Although ConCodeSe only works on Java projects for the purpose of literature comparison, the principles (take the best of various rankings, score class names occurring in the BR higher than other terms, look for class names in particular positions of the summary and of the stack trace if it exists) are applicable to other object-oriented programming languages.

7 Concluding remarks

This paper contributes a novel algorithm that, given a bug report (BR) and the application's source code files, uses a combination of lexical and structural information to suggest, in a ranked order, files that may have to be changed to implement the BR. The algorithm considers words in certain positions of the BR summary and of the stack trace (if available in a bug report) as well as source code comments, stemming, and a combination of both independently, to derive the best rank for each file.

We compared the results to eight existing approaches, using their 5 evaluation criteria and their datasets (4626 BRs from 6 OSS applications), to which we added 39 BRs from 2 other applications. We found that our approach improved the ranking of the affected files, increasing in a statistical significant way the percentage of BRs for which a relevant file is placed among the top-1, 5, 10, which is respectively 44, 69 and 76%, on average. This is an improvement of 23, 16 and 11% respectively over the best performing current state-of-the-art tool.

We also improved, in certain cases substantially, the mean reciprocal rank value for all eight applications evaluated, thereby reducing the number of files to inspect before finding a relevant file. However, our user study, to our knowledge the first on IR-based bug localisation in an industrial setting, confirms results by other authors that developers only tend to look at the first 10 ranked results.

We evaluated the algorithm on four very different industrial applications (one of the two provided by us and three from the user studies), and placed at least one file in the top-10 for 88% of bug reports on average, thus confirming the applicability of our approach also in commercial environments.

Our approach not only outperforms other approaches, it does so in a simpler, faster, and more general way: it uses the least artefacts necessary (one bug report and the source codebase it was reported on), not requiring past information like version history or similar bug reports that have been closed, nor the tuning of any weight factors to combine scores, nor the use of machine learning.

Although our IR-based approach can be also applied to feature requests and not just bug reports, as it does not depend on past BRs, the evaluation datasets only include bug reports. Feature requests necessarily don't include a stack trace and they may not mention specific files. We plan to evaluate our approach with feature requests in future work.

Like previous studies, ours shows that it is challenging to find the files affected by a BR: in spite of our improvements, for larger projects 23% of bugs are not located among the top-10 files. Adding history-based heuristics and learning-to-rank, as proposed by other approaches, will certainly further improve the bug location performance. In order to help develop new search approaches, we will offer in an online companion[13] to this paper the full results as an improved baseline for further bug localisation research.

Acknowledgements We thank Hongyu Zhang for kindly providing BugLocator and its datasets (Zhou et al. 2012), Ripon Saha for the BLUiR dataset (Saha et al. 2013), Laura Moreno for the LOBSTER dataset (Moreno et al. 2014) and Chu Pan Wong for the BRTracer tool source code (Wong et al. 2014). We thank Simon Butler for his assistance in using the JIM tool (Butler et al. 2010), and our industrial partner, a global financial IT solutions provider located in southern Germany, for providing the Pillar2 artefacts and their input on diverse information required. We thank Miguel Goulão for his suggestions and the developers who participated in the user study for their valuable comments. We thank the anonymous reviewers for their insightful comments and additional references, which helped improve the paper.

References

Abebe, S.L., Haiduc, S., Tonella, P., Marcus, A.: The effect of lexicon bad smells on concept location in source code. In: Proceedings of the International Working Conference on Source Code Analysis and Manipulation, pp. 125–134 (2011)

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering traceability links between code and documentation. IEEE Trans. Softw. Eng. **28**(10), 970–983 (2002)

Basel-II: International Convergence of Capital Measurement and Capital Standards: A Revised Framework—Comprehensive Version (2006). http://www.bis.org/publ/bcbs128.htm. Accessed August 2016

Bennett, K.H., Rajlich, V.T.: Software maintenance and evolution: a roadmap. In: Proceedings of the Conference on the Future of Software Engineering, ICSE '00, pp. 73–87. ACM (2000)

[13] http://concodese.com

Bettenburg, N., Premraj, R., Zimmermann, T., Kim, S.: Extracting structural information from bug reports. In: Proceedings of the International Working Conference on Mining Software Repositories, MSR '08, pp. 27–30. ACM (2008)

Boslaugh, S., Watters, P.: Statistics in a Nutshell, 1st edn. O'Reilly Publishing (2008)

Butler, S., Wermelinger, M., Yu, Y., Sharp, H.: Exploring the influence of identifier names on code quality: an empirical study. In: Proceedings of the European Conference on Software Maintenance and Reengineering, pp. 156–165 (2010)

Butler, S., Wermelinger, M., Yu, Y., Sharp, H.: Improving the tokenisation of identifier names. In: Proceedings of the European Conference on Object-Oriented Programming, pp. 130–154. Springer, Berlin, Heidelberg (2011)

Dilshener, T., Wermelinger, M., Yu, Y.: Locating bugs without looking back. In: Proceedings of the International Conference on Mining Software Repositories, MSR '16, pp. 286–290. ACM (2016)

Dilshener, T., Wermelinger, M.: Relating developers' concepts and artefact vocabulary in a financial software module. In: Proceedings of the International Conference on Software Maintenance, pp. 412–417. IEEE (2011)

Eisenbarth, T., Koschke, R., Simon, D.: Aiding program comprehension by static and dynamic feature analysis. In: Proceedings of the International Conference on Software Maintenance, pp. 602–611 (2001)

Gethers, M., Oliveto, R., Poshyvanyk, D., Lucia, A.D.: On integrating orthogonal information retrieval methods to improve traceability recovery. In: Proceedings of the International Conference on Software Maintenance, pp. 133–142. IEEE (2011)

Hill, E., Pollock, L., Vijay-Shanker, K.: Exploring the neighborhood with dora to expedite software maintenance. In: Proceedings of the International Conference on Automated Software Engineering, ASE '07, pp. 14–23. ACM (2007)

Kevic, K., Fritz, T.: Automatic search term identification for change tasks. In: Companion Proceedings of the International Conference on Software Engineering. ICSE Companion 2014, pp. 468–471. ACM (2014)

Kochhar, P.S., Tian, Y., Lo, D.: Potential biases in bug localization: Do they matter? In: Proceedings of the International Conference on Automated Software Engineering, ASE '14, pp. 803–814. ACM (2014)

Kochhar, P.S., Xia, X., Lo, D., Li, S.: Practitioners' expectations on automated fault localization. In: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2016, pp. 165–176. ACM (2016)

Lawrie, D.: Discussion of Appropriate Evaluation Metrics, 1st Workshop on Text Analysis in Software Maintenance (2012). https://dibt.unimol.it/TAinSM2012/slides/dawn.pdf. Accessed April 2016

Le, T.D.B., Oentaryo, R.J., Lo, D.: Information retrieval and spectrum based bug localization: better together. In: Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on Foundations of Software Engineering, ESEC/FSE 2015, pp. 579–590. ACM (2015)

Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., Zhai, C.: Have things changed now?: an empirical study of bug characteristics in modern open source software. In: Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability, ASID '06, pp. 25–33. ACM (2006)

Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press, Cambridge (2008)

Marcus, A., Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Proceedings of the International Conference on Software Engineering, ICSE '03, pp. 125–135. IEEE Computer Society (2003)

Marcus, A., Rajlich, V., Buchta, J., Petrenko, M., Sergeyev, A.: Static techniques for concept location in object-oriented code. In: Proceedings of the International Workshop on Program Comprehension, pp. 33–42 (2005)

Moreno, L., Bandara, W., Haiduc, S., Marcus, A.: On the relationship between the vocabulary of bug reports and source code. In: Proceedings of the International Conference on Software Maintenance, pp. 452–455. IEEE (2013)

Moreno, L., Treadway, J.J., Marcus, A., Shen, W.: On the use of stack traces to improve text retrieval-based bug localization. In: Proceedings of the International Conference on Software Maintenance and Evolution, pp. 151–160. IEEE (2014)

Nichols, B.D.: Augmented bug localization using past bug information. In: Proceedings of the Annual Southeast Regional Conference, ACM SE '10, pp. 61:1–61:6. ACM (2010)

Parnin, C., Orso, A.: Are automated debugging techniques actually helping programmers? In: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA '11, pp. 199–209. ACM (2011)

Petrenko, M., Rajlich, V.: Concept location using program dependencies and information retrieval (DepIR). Inf. Softw. Technol. **55**, 651–659 (2013)

Porter, M.F.: An algorithm for suffix stripping. In: Sparck Jones, K., Willett, P. (eds.) Readings in Information Retrieval, pp. 313–316. Morgan Kaufmann Publishers Inc., San Francisco (1997)

Poshyvanyk, D., Guéhéneuc, Y.G., Marcus, A., Antoniol, G., Rajlich, V.: Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Trans. Softw. Eng. **33**, 420–432 (2007)

Rahman, S., Ganguly, K.K., Sakib, K.: An improved bug localization using structured information retrieval and version history. In: Proceedings of the International Conference on Computer and Information Technology, pp. 190–195 (2015)

Rao, S., Kak, A.: Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In: Proceedings of the Working Conference on Mining Software Repositories, MSR '11, pp. 43–52. ACM (2011)

Ratanotayanon, S., Choi, H.J., Sim, S.E.: My repository runneth over: an empirical study on diversifying data sources to improve feature search. In: Proceedings of the International Conference on Program Comprehension, pp. 206–215 (2010)

Saha, R.K., Lease, M., Khurshid, S., Perry, D.E.: Improving bug localization using structured information retrieval. In: Proceedings of the International Conference on Automated Software Engineering, pp. 345–355 (2013)

Salton, G., Buckley, C.: Term-weighting approaches in automatic text retrieval. Inf. Process. Manag. **24**, 513–523 (1988)

Schröter, A., Bettenburg, N., Premraj, R.: Do stack traces help developers fix bugs? In: Proceedings of the Working Conference on Mining Software Repositories, pp. 118–121 (2010)

Sillito, J., Murphy, G.C., De Volder, K.: Asking and answering questions during a programming change task. IEEE Trans. Softw. Eng. **34**, 434–451 (2008)

Sisman, B., Kak, A.C.: Incorporating version histories in information retrieval based bug localization. In: Proceedings of the Working Conference on Mining Software Repositories, pp. 50–59 (2012)

Starke, J., Luce, C., Sillito, J.: Searching and skimming: an exploratory study. In: Proceedings of the International Conference on Software Maintenance, pp. 157–166 (2009)

Uneno, Y., Mizuno, O., Choi, E.H.: Using a distributed representation of words in localizing relevant files for bug reports. In: Proceedings of the International Conference on Software Quality, Reliability and Security, pp. 183–190 (2016)

Voorhees, E.M.: The trec question answering track. Nat. Lang. Eng. **7**, 361–378 (2001)

Wang, S., Lo, D., Lawall, J.: Compositional vector space models for improved bug localization. In: Proceedings of the International Conference on International Conference on Software Maintenance and Evolution, pp. 171–180. IEEE (2014)

Wang, S., Lo, D.: Version history, similar report, and structure: Putting them together for improved bug localization. In: Proceedings of the International Conference on Program Comprehension, ICPC 2014, pp. 53–63. ACM (2014)

Wang, Q., Parnin, C., Orso, A.: Evaluating the usefulness of IR-based fault localization techniques. In: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2015, pp. 1–11. ACM (2015)

Wilde, N., Scully, M.C.: Software reconnaissance: mapping program features to code. J. Softw. Maint. Res. Pract. **7**, 49–62 (1995)

Wong, C.P., Xiong, Y., Zhang, H., Hao, D., Zhang, L., Mei, H.: Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: Proceedings of the International Conference on Software Maintenance and Evolution, pp. 181–190. IEEE (2014)

Xia, X., Bao, L., Lo, D., Li, S.: Automated debugging considered harmful—considered harmful: a user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In: Proceedings of the International Conference on Software Maintenance and Evolution, pp. 267–278. IEEE (2016)

Ye, X., Bunescu, R., Liu, C.: Learning to rank relevant files for bug reports using domain knowledge. In: Proceedings of the International Symposium on Foundations of Software Engineering, FSE 2014, pp. 689–699. ACM (2014)

Ye, X., Shen, H., Ma, X., Bunescu, R., Liu, C.: From word embeddings to document similarities for improved information retrieval in software engineering. In: Proceedings of the International Conference on Software Engineering. ICSE '16, pp. 404–415. ACM (2016)

Youm, K.C., Ahn, J., Kim, J., Lee, E.: Bug localization based on code change histories and bug reports. In: Proceedings of the Asia-Pacific Software Engineering Conference, pp. 190–197 (2015)

Zhou, J., Zhang, H., Lo, D.: Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the International Conference on Software Engineering, ICSE '12, pp. 14–24. IEEE (2012)

Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schröter, A., Weiss, C.: What makes a good bug report? IEEE Trans. Softw. Eng. **36**(5), 618–643 (2010)