Mapping Bug Reports to Relevant Files: A Ranking Model, a Fine-grained Benchmark, and Feature Evaluation

Xin Ye, *Student Member, IEEE,* Razvan Bunescu, and Chang Liu, *Senior Member, IEEE*

Abstract—When a new bug report is received, developers usually need to reproduce the bug and perform code reviews to find the cause, a process that can be tedious and time consuming. A tool for ranking all the source files with respect to how likely they are to contain the cause of the bug would enable developers to narrow down their search and improve productivity. This paper introduces an adaptive ranking approach that leverages project knowledge through functional decomposition of source code, API descriptions of library components, the bug-fixing history, the code change history, and the file dependency graph. Given a bug report, the ranking score of each source file is computed as a weighted combination of an array of features, where the weights are trained automatically on previously solved bug reports using a learning-to-rank technique. We evaluate the ranking system on six large scale open source Java projects, using the before-fix version of the project for every bug report. The experimental results show that the learning-to-rank approach outperforms three recent state-of-the-art methods. In particular, our method makes correct recommendations within the top 10 ranked source files for over 70% of the bug reports in the Eclipse Platform and Tomcat projects.

Index Terms—Bug reports, software maintenance, learning to rank

1 Introduction

A software *bug* or *defect* is a coding mistake that may cause an unintended or unexpected behavior of the software component [12]. Upon discovering an abnormal behavior of the software project, a developer or a user will report it in a document, called a *bug report* or *issue report*. A bug report provides information that could help in fixing a bug, with the overall aim of improving the software quality. A large number of bug reports could be opened during the development life-cycle of a software product. For example, there were 3,389 bug reports created for the Eclipse Platform product in 2013 alone. In a software team, bug reports are extensively used by both managers and developers in their daily development process [15].

A developer who is assigned a bug report usually needs to reproduce the abnormal behavior [35] and perform code reviews [4] in order to find the cause. However, the diversity and uneven quality of bug reports can make this process nontrivial. Essential information is often missing from a bug report [11]. Bacchelli and Bird [4] surveyed 165 managers and 873 programmers, and reported that finding defects requires a high level understanding of the code and familiarity with the relevant source code files. In the survey, 798 respondents answered that it takes time to review unfamiliar files. While the number of source files in a project is usually large, the number of files that contain the bug is usually very small. Therefore, we believe that an automatic approach that ranked the source files with respect to their relevance for the bug report could speed up the bug finding process by narrowing the search to a smaller number of possibly unfamiliar files.

If the bug report is construed as a query and the source code files in the software repository are viewed as a collection of documents, then the problem of finding source files that are relevant for a given bug report can be modeled as a standard task in information retrieval (IR) [41]. As such, we propose to approach it as a ranking problem, in which the source files (documents) are ranked with respect to their *relevance* to a given bug report (query). In this context, relevance is equated with the likelihood that a particular source file contains the cause of the bug described in the bug report. The ranking function is defined as a weighted combination of features, where the features draw heavily on knowledge specific to the software engineering domain in order to measure relevant relationships between the bug report and the source code file. While a bug report may share textual tokens with its relevant source files, in general there is a significant inherent mismatch between the natural language employed in the bug report and the programming language used in the code [7]. Ranking methods that are based on simple lexical matching scores have suboptimal performance, in part due to lexical mismatches between natural language statements in bug reports and technical terms in software systems. Our system contains features that bridge the corresponding *lexical gap* by using project specific API documentation to connect natural language terms in the bug report with programming language constructs in the code. Further-

- *X. Ye, R. Bunescu, and C. Liu are with the School of Electrical Engineering and Computer Science, Ohio University, Athens, OH, 45701.*
E-mail: {xy348709, bunescu, liuc}@ohio.edu

more, source code files may contain a large number of methods of which only a small number may be causing the bug. Correspondingly, the source code is syntactically parsed into methods and the features are designed to exploit method-level measures of relevance for a bug report. It has been previously observed that software *process metrics* (e.g., change history) are more important than *code metrics* (e.g., size of codes) in detecting defects [59]. Consequently, we use the change history of source code as a strong signal for linking fault-prone files with bug reports. Another useful domain specific observation is that a buggy source file may cause more than one abnormal behavior, and therefore may be responsible for similar bug reports. If we equate a bug report with a *user* and a source code file with an *item* that the user may like or not, then we can draw an analogy with recommender systems [46] and employ the concept of *collaborative filtering*. Thus, if previously fixed bug reports are textually similar with the current bug report, then files that have been associated with the similar reports may also be relevant for the current report. We expect complex code to be more prone to bugs than simple code. Correspondingly, we design query-independent features that capture the code complexity through proxy properties derived from the file dependency graph, such as the PageRank [54] score of a source file or the number of file dependencies.

The resulting ranking function is a linear combination of features, whose weights are automatically trained on previously solved bug reports using a learning-to-rank technique. We have conducted extensive empirical evaluations on six large-scale, open-source software projects with more than 22,000 bug reports in total. To avoid contaminating the training data with future bug-fixing information from previous reports, we created fine-grained benchmarks by checking out the before-fix version of the project for every bug report. Experimental results on the before-fix versions show that our system significantly outperforms a number of strong baselines as well as three recent state-of-the-art approaches. In particular, when evaluated on the Eclipse Platform UI dataset containing over 6,400 solved bug reports, the learning-to-rank system is able to successfully locate the true buggy files within the top 10 recommendations for over 70% of the bug reports, corresponding to a mean average precision of over 40%. Overall, we see our adaptive ranking approach as being generally applicable to software projects for which a sufficient amount of project specific knowledge, in the form of version control history, bug-fixing history, API documentation, and syntactically parsed code, is readily available.

The main contributions of this paper include: a ranking approach to the problem of mapping source files to bug reports that enables the seamless integration of a wide diversity of features; exploiting previously fixed bug reports as training examples for the proposed ranking model in conjunction with a learning-to-rank technique; using the file dependency graph to define features that capture a measure of code complexity; fine-grained benchmark datasets created by checking out a before-fix version of the source code package for each bug report; extensive evaluation and comparisons with existing state-of-the-art methods; and a thorough evaluation of the impact that features have on the ranking accuracy.

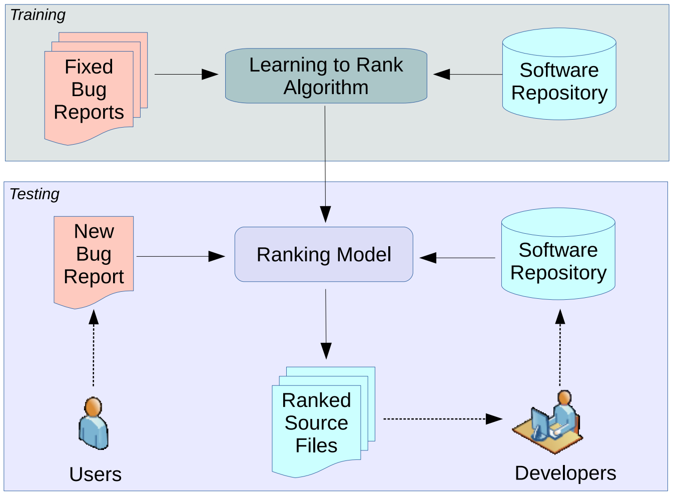


Fig. 1. System architecture for *training* and *testing*.

The rest of the paper is structured as follows. Section 2 outlines the system architecture. This is followed in Section 3 by a detailed description of the features employed in the definition of the ranking function. The fine-grained benchmark datasets are introduced in Section 4, followed by a description of the experimental evaluation setting and results in Sections 5 to 7. Section 8 describes the results of a feature selection procedure, followed by experiments that are intended to elucidate the importance of every feature in the final system performance. After a discussion of related work in Section 10, the paper ends with future work and concluding remarks.

2 Ranking Model

Figure 1 shows the high-level architecture of the system. A ranking model is trained to compute a matching score for any bug report r and source code file s combination. The scoring function $f(r, s)$ is defined as a weighted sum of k features, where each feature $\phi_i(r, s)$ measures a specific relationship between the source file s and the received bug report r:

$$f(r, s) = \mathbf{w}^T \Phi(r, s) = \sum_{i=1}^{k} w_i * \phi_i(r, s) \tag{1}$$

Given an arbitrary bug report r as input at test time, the model computes the score $f(r, s)$ for each source file s in the software project and uses this value to rank all the files in descending order. The user is then presented with a ranked list of files, with the expectation that files appearing higher in the list are more likely to be relevant for the bug report i.e., more likely to contain the cause of the bug.

The model parameters w_i are trained on previously solved bug reports using a learning-to-rank technique. In this learning framework, the optimization procedure tries to find a set of parameters for which the scoring function ranks the files that are known to be relevant for a bug report at the top of the list for that bug report.

3 Feature Representation

The proposed ranking model requires that a bug report - source file pair (r, s) be represented as a vector of k features $\Phi(r, s) = [\phi_i(r, s)]_{1 \leq i \leq k}$. The overall set of 19 features used in the ranking model is summarized in Table 1. As shown in the last column in the table, we distinguish between two major categories of features:

- **Query Dependent**: These are features $\phi_i(r, s)$ that depend on both the bug report r and the source code file s. A query dependent feature represents a specific relationship between the bug report and the source file, and thus may be useful in determining directly whether the source code file s contains a bug that is relevant for the bug report r.
- **Query Independent**: These are features that depend only on the source code file i.e., their computation does not require knowledge of the bug report query. As such, query independent features may be used to estimate the likelihood that a source code file contains a bug, irrespective of the bug report.

We hypothesize that both types of features are useful when combined in an overall ranking model. Experimental results from Section 8 on feature selection confirm the utility of both types of features, although the impact of some of the features will be shown to be project dependent.

The remaining of this section describes in detail the 19 features shown in Table 1. Features ϕ_1 to ϕ_6 have been originally introduced by us in [70], whereas features ϕ_7 to ϕ_{14} are adapted from the structural IR approach of Saha et al. [62]. The newly proposed features ϕ_{15} to ϕ_{19} are query-independent features that exploit properties of the file dependency graph deemed relevant for detecting files that are likely to contain bugs.

3.1 Vector Space Representation

If we regard the bug report as a query and the source code file as a text document, then we can employ the classic Vector Space Model (VSM) for ranking, a standard model used in information retrieval. In this model, both the query and the document are represented as vectors of term weights. Given an arbitrary document d (a bug report or a source code file), we compute the term weights $w_{t,d}$ for each term t in the vocabulary based on the classical *tf.idf* weighting scheme in which the term frequency factors are normalized, as follows:

$$\begin{aligned} w_{t,d} &= nf_{t,d} \times idf_t \\ nf_{t,d} &= 0.5 + \frac{0.5 \times tf_{t,d}}{\max_{t \in d} tf_{t,d}} \quad idf_t = log\frac{N}{df_t} \end{aligned} \tag{2}$$

The *term frequency* factor $tf_{t,d}$ represents the number of occurrences of term t in document d, whereas the *document frequency* factor df_t represents the number of documents in the repository that contain term t. N is to the total number of documents in the repository, while idf_t refers to the *inverse document frequency*, which is computed using a logarithm in order to dampen the effect of the document frequency factor in the overall term weight.

3.1.1 Surface Lexical Similarity

For a bug report, we use both its summary and description to create the VSM representation. For a source file, we use its whole content – code and comments. To tokenize an input document, we first split the text into a bag of words using white spaces. We then remove punctuation, numbers, and standard IR stop words such as conjunctions or determiners. Compound words such as "WorkBench" are split into their components based on capital letters, although more sophisticated methods such as [23], [62] could have been used here too. The bag of words representation of the document is then augmented with the resulting tokens – "Work" and "Bench" in this example – while also keeping the original word as a token. Finally, all words are reduced to their stem using the Porter stemmer, as implemented in the NLTK[1] package. This process will reduce derivationally related words such as "programming" and "programs" to the same stem "program", which is known to have a positive impact on the recall performance of the final system.

Let V be the vocabulary of all text tokens appearing in bug reports and source code files. Let $\mathbf{r} = [w_{t,r} | t \in V]$ and $\mathbf{s} = [w_{t,s} | t \in V]$ be the VSM vector representations of the bug report r and the source code file s, where the term weights $w_{t,r}$ and $w_{t,s}$ are computed using the *tf.idf* formula as shown in Equation 2 above. Once the vector space representations are computed, the textual similarity between a source code file and a bug report can be computed using the standard *cosine similarity* between their corresponding vectors:

$$sim(r, s) = cos(\mathbf{r}, \mathbf{s}) = \frac{\mathbf{r}^T \mathbf{s}}{\|\mathbf{r}\| \|\mathbf{s}\|} \tag{3}$$

This is simply the inner product of the two vectors, normalized by their Euclidean norm.

The VSM cosine similarity could be used directly as a feature in the computation of the scoring function in Equation 1. However, this would ignore the fact that bugs are often localized in a small portion of the code, such as one method. When the source file is large,

[1] http://www.nltk.org/api/nltk.stem.html

TABLE 1
Features used in the ranking model (*Yes*† *means the features depend on the bug report only through its timestamp*).

Feature	Section	Short Description	Formula	Q-Dependent?
ϕ_1	3.1.1	Surface lexical similarity	$\phi_1(r,s) = \max(\{sim(r,s)\} \cup \{sim(r,m) \mid m \in s\})$	Yes
ϕ_2	3.1.2	API-enriched lexical similarity	$\phi_2(r,s) = \max\{sim(r,s.api)\} \cup \{sim(r,m.api) \mid m \in s\}$	Yes
ϕ_3	3.2	Collaborative filtering score	$\phi_3(r,s) = sim(r, R(s))$	Yes
ϕ_4	3.3	Class name similarity	$\phi_4(r,s) = \lvert s.class \rvert * \mathbf{1}[s.class \in r.summary]$	Yes
ϕ_5	3.4.1	Bug-fixing recency	$\phi_5(r,s) = (r.month - last(r,s).month + 1)^{-1}$	Yes†
ϕ_6	3.4.2	Bug-fixing frequency	$\phi_6(r,s) = \lvert br(r,s) \rvert$	Yes†
ϕ_7	3.5	Summary-class names similarity	$\phi_7(r,s) = sim(r.summary, s.class))$	Yes
ϕ_8	3.5	Summary-method names similarity	$\phi_8(r,s) = sim(r.summary, s.method))$	Yes
ϕ_9	3.5	Summary-variable names similarity	$\phi_9(r,s) = sim(r.summary, s.variable))$	Yes
ϕ_{10}	3.5	Summary-comments similarity	$\phi_{10}(r,s) = sim(r.summary, s.comment))$	Yes
ϕ_{11}	3.5	Description-class names similarity	$\phi_{11}(r,s) = sim(r.description, s.class))$	Yes
ϕ_{12}	3.5	Description-method names similarity	$\phi_{12}(r,s) = sim(r.description, s.method))$	Yes
ϕ_{13}	3.5	Description-variable names similarity	$\phi_{13}(r,s) = sim(r.description, s.variable))$	Yes
ϕ_{14}	3.5	Description-comments similarity	$\phi_{14}(r,s) = sim(r.description, s.comment))$	Yes
ϕ_{15}	3.6.1	In-links = # of file dependencies of s	$\phi_{15}(r,s) = s.inLinks$	No
ϕ_{16}	3.6.1	Out-links = # of files that depend on s	$\phi_{16}(r,s) = s.outLinks$	No
ϕ_{17}	3.6.2	PageRank score	$\phi_{17}(r,s) = PageRank(s)$	No
ϕ_{18}	3.6.3	Authority score	$\phi_{18}(r,s) = Authority(s)$	No
ϕ_{19}	3.6.3	Hub score	$\phi_{19}(r,s) = Hub(s)$	No

its corresponding norm will also be large, which will result in a small cosine similarity with the bug report, even though one method in the file may be actually very relevant for the same bug report. Therefore, we use the AST parser from Eclipse JDT[2] and segment the source code into methods in order to compute per-method similarities with the bug report. We consider each method m as a separate document and calculate its lexical similarity with the bug report using the same cosine similarity formula. We then compute a surface lexical similarity feature as follows:

$$\phi_1(r,s) = \max(\{sim(r,s)\} \cup \{sim(r,m) \mid m \in s\}) \tag{4}$$

i.e., the maximum from all per-method similarities and the whole file similarity. This is obviously a query-dependent feature.

3.1.2 API-Enriched Lexical Similarity

In general, most of the text in a bug report is expressed in natural language (e.g., English), whereas most of the content of a source code file is expressed in a programming language (e.g., Java). Since the inner product used in the cosine similarity function has non-zero terms only for tokens that are in common between the bug report and the source file, this implies that the surface lexical similarity feature described in the previous section will be helpful only when 1) the source code has extensive, comprehensive comments, or 2) the bug report includes snippets of code or programming language constructs such as names of classes or methods. In practice, it is often the case that the bug report and a relevant buggy file share very few tokens, if any. For example, Figure 2 below shows a sample from a bug report[3] from the Eclipse project. This bug report describes a defect in which the toolbar is missing icons and showing wrong menus. Figure 3 shows a snippet from a buggy file that is known to be relevant for this report. At the surface level, the two documents do not share any tokens, consequently their cosine similarity will be 0, thus useless for determining relevance.

Bug ID: 339286
Summary: Toolbars missing icons and show wrong menus.
Description: The toolbars for my stacked views were: missing icons, showing the wrong drop-down menus (from others in the stack), showing multiple drop-down menus, missing the min/max buttons ...

Fig. 2. Eclipse bug report 339286.

```
public class PartRenderingEngine
                    implements IPresentationEngine {
private EventHandler trimHandler = new EventHandler()
{
  public void handleEvent(Event event) { ...
    MTrimmedWindow window =
                    (MTrimmedWindow) changedObj;
    ... } ... } ... }
```

Fig. 3. Code from PartRenderingEngine.java

However, we can bridge the lexical gap by using the API specification of the classes and interfaces used in the source code. The buggy file PartRenderingEngine.java declares a variable *window* whose type is `MTrimmedWindow`. As specified in the Eclipse API, `MUILabel` is a super-interface of `MTrimmedWindow`. As can be seen in Figure 4, the API documentation of the

[2] http://www.eclipse.org/jdt/
[3] https://bugs.eclipse.org/bugs/show_bug.cgi?id=339286

Interface `MUILabel`
All Known Subinterfaces: `MTrimmedWindow`, ...
Description: A representation of the model object 'UI Label'. This is a mix in that will be used for UI Elements that are capable of showing label information in the GUI (e.g. Parts, Menus / Toolbars, Perspectives, ...). The following features are supported: Label, Icon URI, Tooltip ...

Fig. 4. API specification for `MUILabel` interface.

`MUILabel` interface mentions tokens such as *toolbar*, *icon*, and *menu* that also appear in the bug report.

Therefore, for each method in a source file, we extract a set of class and interface names from the explicit type declarations of all local variables. Using the project API specification, we obtain the textual descriptions of these classes and interfaces, including the descriptions of all their direct or indirect super-classes or super-interfaces. For each method m we create a document $m.api$ by concatenating the corresponding API descriptions. Finally, we take the API specifications of all methods in the source file s and concatenate them into an overall document $s.api = \cup_{m \in s} m.api$. We then compute an API-enriched lexical similarity feature as follows:

$$\phi_2(r, s) = \max\{sim(r, s.api)\} \cup \{sim(r, m.api) | m \in s\} \tag{5}$$

i.e., the maximum from all per-method API similarities and the whole file API similarity. This is a query-dependent feature.

Bug ID: 378535
Summary: "Close All" and "Close Others" menu options available when right clicking on tab in PartStack when no part is closeable.
Description: If I create a PartStack that contains multiple parts but none of the parts are closeable, when I right click on any of the tabs I get menu options for "Close All" and "Close Others". Selection of either of the menu options doesn't cause any tabs to be closed since none of the tabs can be closed. I don't think the menu options should be available if none of the tabs can be closed ...

Fig. 5. Eclipse bug report 378535.

Bug ID: 329950
Summary: "Close All" and "Close Others" may cause bundle activation.
Description: ...

Bug ID: 325722
Summary: "Close"-related context menu actions should show up for all stacks and apply to all items.
Description: ...

Bug ID: 313328
Summary: Close parts under stacks with middle mouse click.
Description: ...

Fig. 6. Bug reports that are similar with 378535.

3.2 Collaborative Filtering Score

It has been observed in [49] that a file that has been fixed before may be responsible for similar bugs. For example, Figure 5 displays an Eclipse bug report about incorrect menu options for parts that are not closeable. Figure 6 shows three other bug reports that were solved before bug 378535 was reported. These three reports describe similar defects and therefore share many keywords with report 378535 (shown underlined in the figures). Consequently, it is not surprising that source file *StackRenderer.java*, which had been previously found to be relevant for the three reports in Figure 5, was also found to be relevant for the textually similar bug report in Figure 6.

This *collaborative filtering* effect has been used before in other domains to improve the accuracy of recommender systems [46], consequently it is expected to be beneficial in our retrieval setting, too. Given a bug report r and a source code file s, let $br(r, s)$ be the set of bug reports for which file s was fixed before r was reported. The collaborative filtering feature is then defined as follows:

$$\phi_3(r, s) = sim(r, br(r, s)) \tag{6}$$

The feature computes the textual similarity between the text of the current bug report r and the summaries of all the bug reports in $br(r, s)$. This feature is query-dependent.

3.3 Class Name Similarity

A bug report may directly mention a class name in the summary, which provides a useful signal that the corresponding source file implementing that class may be relevant for the bug report. Our hypothesis is that the signal becomes stronger when the class name is longer and thus more specific. For example, the summary of the Eclipse bug report 409274 contains the class names `WorkbenchWindow`, `Workbench`, and `Window` after tokenization, but only *WorkbenchWindow.java* is a relevant file.

Let $s.class$ denote the name of the main class implemented in source file s, and $|s.class|$ the name length. Based on the observation above, we define a class name similarity feature as follows:

$$\phi_4(r, s) = \begin{cases} |s.class| & \text{if } s.class \in r \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

This is a query-dependent feature. This feature will be automatically normalized during the feature scaling step described in Section 3.7.

3.4 File Revision History

The source code change history provides information that can help predict fault-prone files [59]. For example, a source code file that was fixed very recently is more likely to still contain bugs than a file that was last fixed long time in the past, or never fixed.

3.4.1 Bug-Fixing Recency

As in Section 3.2, let $br(r, s)$ be the set of bug reports for which file s was fixed before bug report r was created. Let $last(r, s) \in br(r, s)$ be the most recent previously fixed bug. Also, for any bug report r, let $r.month$ denote the month when the bug report was created. We then define the bug-fixing recency feature to be the inverse of the distance in months between r and $last(r, s)$:

$$\phi_5(r, s) = (r.month - last(r, s).month + 1)^{-1} \tag{8}$$

Thus, if s was last fixed in the same month that r was created, $\phi_5(r, s)$ is 1. If s was last fixed one month before r was created, $\phi_5(r, s)$ is 0.5.

3.4.2 Bug-Fixing Frequency

A source file that has been frequently fixed may be a fault-prone file. Consequently, we define a bug-fixing frequency feature as the number of times a source file has been fixed before the current bug report:

$$\phi_6(r, s) = |br(r, s)| \tag{9}$$

This feature will be automatically normalized during the feature scaling step described in Section 3.7 below.

Neither of the two features $\phi_5(r, s)$ or $\phi_6(r, s)$ rely on the content of the bug report r. However, their calculation still depends on the timestamp of the bug report, therefore we consider the two revision history features as query-dependent.

3.5 Structural Information Retrieval

By computing similarities with each method and then maximizing across all methods in a source file, feature ϕ_1 alleviates the problem of the small similarities that result for localized bugs, when using a straightforward cosine similarity formula in which the normalization factor is correlated with the length of the file. A related problem may occur when the bug report is very similar with a particular type of content from a source file (e.g. comments, method names, or class names) and dissimilar with everything else, yet the cosine similarity with the entire file is very small due to its large size. To model such cases, we follow the structural IR approach of Saha et al. [62], in which a source code file s is parsed into four document fields: all class names in $s.class$, all methods names in $s.method$, all variable names in $s.variable$, and all comments in $s.comment$. For example, a field such as $s.method$ is equivalent with a document that contains all the method names defined in the source code file s. The summary and the description of a bug report r are used to create two query fields: $r.summary$ and $r.description$, respectively.

The report-based bug localization approach from [62] computes an overall ranking score as the simple sum of the lexical similarities of all possible 8 document-query field pairs. Instead of using a simple sum, we propose that the eight similarities are combined as separate features into the overall ranking function, as shown in the weighted sum from Equation 1, in which the weights are learned automatically from training data.

The 8 features ϕ_7 to ϕ_{14} can be seen as fine grained versions of feature ϕ_1, in which the input fields are the source file and the bug report in their entirety.

$$\phi_7(r, s) = sim(r.summary, s.class)) \tag{10}$$
$$\phi_8(r, s) = sim(r.summary, s.method)) \tag{11}$$
$$\phi_9(r, s) = sim(r.summary, s.variable)) \tag{12}$$
$$\phi_{10}(r, s) = sim(r.summary, s.comment)) \tag{13}$$
$$\phi_{11}(r, s) = sim(r.description, s.class)) \tag{14}$$
$$\phi_{12}(r, s) = sim(r.description, s.method)) \tag{15}$$
$$\phi_{13}(r, s) = sim(r.description, s.variable)) \tag{16}$$
$$\phi_{14}(r, s) = sim(r.description, s.comment)) \tag{17}$$

All eight features are query-dependent.

3.6 The File Dependency Graph

We expect complex code to be more prone to bugs than simple code. Thus, the complexity of the source code contained in a file can provide another useful signal with respect to the likelihood that the file contains bugs. An accurate measure of code complexity would require a good representation of the semantics of the code. Since a comprehensive semantic analysis of code is currently not feasible, we resort to a characterization of code complexity based on syntactic features. For example, a proxy measure for the complexity of a source code file can be defined such that:

1) The complexity increases with every new class (or more generally, code construct) that is used in the code. Since each class can be mapped to a particular source code file that implements it, we can reformulate this property and say that the complexity of a source code file s is positively correlated with the number of source code files on which s depends i.e., the number of file dependencies of s.
2) The complexity of a source code file s depends not only on the number of file dependencies, but also on the actual complexity of each file dependency. If s depends on two other source files s_1 and s_2, and s_1 (the class implemented therein) is more complex than s_2, we expect that the use of s_1 by s is more likely to lead to bugs than the use of s_2. That is to say, using a complex construct is more difficult than using a simple construct, and therefore more likely to lead to bugs.
3) The perceived complexity of a code artifact (class, source code file) decreases with each additional

use, as programmers become more familiar with it and thus less likely to use it incorrectly.

4) The source code file complexity depends also on factors other than the complexity of each of its file dependencies. This is a catch-all component of the complexity measure that, although difficult to fully capture formally, needs to be addressed in any useful operational definition of code complexity.

The first three properties above lead to a recursive definition of complexity that resembles the definition of web page quality used in the PageRank algorithm [54], where a hyperlink from page p_1 to page p_2 confers to p_2 a fraction of the quality of p_1, based on the assumption that, by linking to p_2, the creator of p_1 thinks that p_2 is of high quality.

Let $G = (E, V)$ be the file dependency graph of a given project, where V is the set of source code files contained in the project. An edge $t \rightarrow s \in E$ indicates that source file t is used in source file s i.e., t is a file dependency of s. Furthermore, let $s.inLinks$ and $s.outLinks$ denote the number of edges entering s, or leaving from s, respectively. Then the PageRank complexity of a source code file s has the following form:

$$K(s) = \sum_{t \rightarrow s} \alpha \times \frac{K(t)}{|t.outLinks|} + (1 - \alpha) \times \frac{1}{|V|}. \tag{18}$$

The first term captures the first three properties enumerated above: 1) each file dependency t increases the complexity of s; 2) a source file t's contribution to the complexity of s depends on the complexity of t itself; 3) t's complexity contribution decreases with the number of files using it. The second term captures the fourth component of the complexity: 4) a fraction $(1 - \alpha)$ of the overall complexity of s is due to factors not captured through file dependencies. For simplicity and tractability, we assume that these alternative sources of complexity are distributed uniformly over all the source code files in a project. The damping factor α represent the percentage of complexity that is expected to be captured through file dependencies.

To create the file dependency graph, we extract the file dependency relationships among all Java files within a project using the Eclipse JDT ASTParser. For any given source code file, we infer the file dependencies based on the following node types extracted by the ASTParser: *ImportDeclaration*, *SingleVariableDeclaration*, *MethodInvocation*, *FieldDeclaration*, *ClassInstanceCreation*, and *SimpleType*. For example, Figure 7 below shows a code snippet from the the source file *ResourceRegistryKeyFactory.java*, in the project Eclipse Platform UI, version 5da5952. When run on this code, the ASTParser will extract the following dependencies: `CSSValue` from *ImportDeclaration* and *SingleVariableDeclaration*, and `CSSResourcesHelpers` from *MethodInvocation*. Therefore, we infer that *ResourceRegistryKeyFactory.java* depends on the corresponding Java files *CSSValue.java* and *CSSResourcesHelpers.java*.

```
package org.eclipse.e4.ui.css.core.resources;

import org.w3c.dom.css.CSSValue;

public class ResourceRegistryKeyFactory {
 public Object createKey(CSSValue value) {
 return CSSResourcesHelpers.getCSSValueKey(value);
 }
}
```

Fig. 7. Code from ResourceRegistryKeyFactory.java

We use the above steps on every source file in the project, and find that *SWTResourceRegistryKeyFactory.java* and *AbstractCSSEngine.java* are two files that depend on *ResourceRegistryKeyFactory.java*. The resulting set of four file dependencies are illustrated in Figure 8 below, which shows a subgraph of the overall dependency graph.

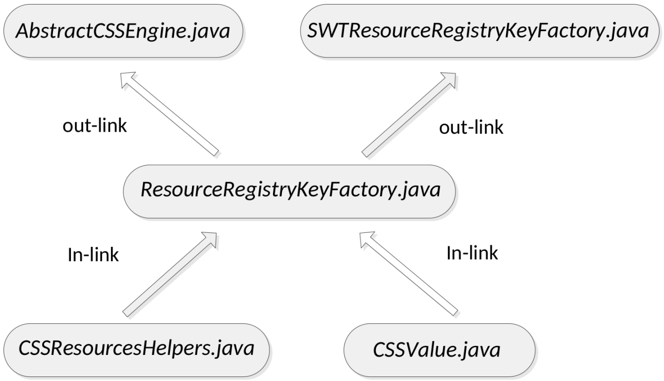


Fig. 8. Dependency Graph for *ResourceRegistryKeyFactory.java*

Based on the dependency graph of the project, we define the five graph features below.

3.6.1 Local Graph Features

Given a source code file s, we compute the number of file dependencies of s (the number of in-links of s) as well as the number of files that depend on s (the number of out-links of s):

$$\phi_{15}(r, s) = s.inLinks \tag{19}$$

$$\phi_{16}(r, s) = s.outLinks \tag{20}$$

3.6.2 Complexity Score as PageRank

We model the complexity of a source code file using the PageRank approach described above:

$$\phi_{17}(r, s) = K(s) = PageRank(s) \tag{21}$$

We use the Java Universal Network/Graph Framework (JUNG[4]) to calculate the PageRank of each source code file in the project dependency graph.

[4] http://jung.sourceforge.net/

3.6.3 Hubs and Authorities Scores

The Hubs and Authorities [33], also known as Hyperlink-Induced Topic Search (HITS), is a link-based algorithm for finding web pages with high quality information on a topic and web pages with high quality reference links for a topic. A web page with high authority score is recognized as an expert that provides significant information on a topic, and is thus linked by many hub pages. A web page with a high hub score is recognized as a good list of links to many authority pages. In a Java project, an abstract class or an interface that is extended or implemented by many other files is expected to have a high hub score. Similarly, a source code file that contains the implementation of a class that extends another class and implements many interfaces is expected to have a high authority score.

$$\phi_{18}(r,s) = Hub(s) \tag{22}$$

$$\phi_{19}(r,s) = Authority(s) \tag{23}$$

We use JUNG to calculate the hubs and authorities scores of each source code file in the project dependency graph.

All five dependency features are query-independent.

3.7 Feature Scaling

Features with widely different ranges of values are detrimental in machine learning models. Feature scaling helps bring all features to the same scale so that they become comparable with each other. For an arbitrary feature ϕ, let $\phi.min$ and $\phi.max$ be the minimum and the maximum observed values in the training dataset. A feature ϕ may have values in the testing dataset that are larger than $\phi.max$, or smaller than $\phi.min$. Therefore, examples in both the training and testing dataset will have their features scaled as follows:

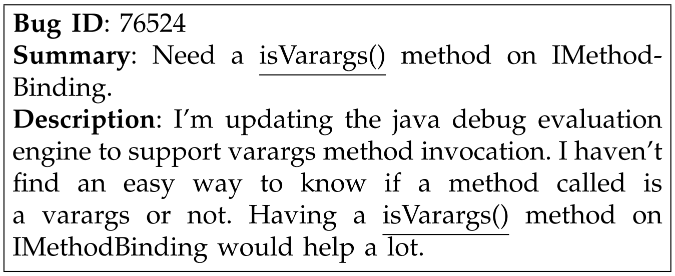
$$\phi' = \begin{cases} 0 & \text{if } \phi < \phi.min \\ \dfrac{\phi - \phi.min}{\phi.max - \phi.min} & \text{if } \phi.min \leq \phi \leq \phi.max \\ 1 & \text{if } \phi > \phi.max \end{cases} \tag{24}$$

4 Fine-grained Benchmark Datasets

Previous approaches to bug localization used just one code revision to evaluate the system performance on multiple bug reports. However, software bugs are often found in different revisions of the source code package. Using a fixed revision during evaluation is problematic for at least two major reasons:

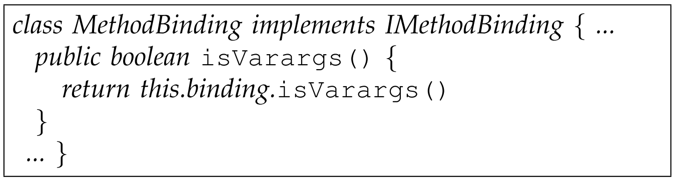
1) The fixed revision that is used for evaluation may contain future bug-fixing information for older bug reports.
2) A relevant buggy file might not even exist in the fixed revision, if it were deleted after the bug was reported.

Consequently, using just one revision of the source code package for evaluation may lead to performance assessments that overestimate the actual performance of the system when used in practice.



Bug ID: 76524
Summary: Need a isVarargs() method on IMethodBinding.
Description: I'm updating the java debug evaluation engine to support varargs method invocation. I haven't find an easy way to know if a method called is a varargs or not. Having a isVarargs() method on IMethodBinding would help a lot.

Fig. 9. Eclipse bug report 76524.



```
class MethodBinding implements IMethodBinding { ...
  public boolean isVarargs() {
    return this.binding.isVarargs()
  }
 ... }
```

Fig. 10. Code snippet from MethodBinding.java.

For example, the dataset from [62], [71] associates 3,075 bug reports with a fixed version of the Eclipse 3.1 source code package[5]. Figure 9 shows a bug report in which the author recommends adding a method called `isVarargs()`. One of the files that were fixed for this bug report is `MethodBinding.java`. At the time the bug report was submitted, this class did not contain a `isVarargs()` method. However, the fixed revision of Eclipse 3.1 used in the dataset contains a future version of `MethodBinding.java` in which the method `isVarargs()` has already been added, as shown in Figure 10. The presence of future bug-fixing information in the fixed revision dataset is likely to lead to an unrealistic estimate of the system performance, as the bug report has a larger textual similarity with the future version of the `MethodBinding.java` file than with the current version (the version at the time when the bug report was submitted).

Furthermore, a relevant buggy file that is available at the time the bug report is submitted may be deleted from a future revision. If that future revision is used for evaluation, the relevant buggy file will not be considered at all during evaluation, which will again result in a flawed estimate of system performance. For example, the same dataset introduced in [62], [71] associates 286 bug reports with a fixed version of the AspectJ project[6]. For bug 107299, `ConfigParser.java`[7] is a relevant buggy file. However, this file was deleted between the time the bug report was submitted and the time of the future revision of AspectJ that was used in the dataset.

A somewhat smaller problem with the dataset above is caused by the decision to use the package name plus the class name to indicate a file that was fixed for a bug report. This may lead to ambiguities, since

[5] http://archive.eclipse.org/eclipse/downloads/drops/R-3.1-200506271435/

[6] https://code.google.com/p/bugcenter

[7] util/src/org/aspectj/util/ConfigParser.java

two or more files may match that same package and class name. For example, the dataset specifies that *org.eclipse.core.runtime.IAdaptable* is a file that was fixed for the Eclipse bug 88850. However, this name matches two different files in the Eclipse project: *plugins/org.eclipse.core.runtime/src/org/eclipse/core/runtime/I-Adaptable.java* and *plugins/org.eclipse.jface/runtime-util /org/eclipse/core/runtime/IAdaptable.java*. Whenever this happens, it is unclear which file to use during evaluation.

There are feature locations benchmarks, such as the one proposed by Dit et al. [20], that suffer from the same major issue identified above: a fixed version of the course code package is used for evaluation with multiple bug reports. For example, the dataset in [20] contains the source code of jEdit 4.3 whereas the bug reports refer to snapshots of the code between versions 4.2 and 4.3. Furthermore, there are in total only 593 bug reports from 6 Java projects, which would be insufficient for both training and evaluating a project specific ranking model.

To avoid the problems associated with using a fixed code revision, we create a *fine-grained benchmark* dataset for each project by checking out a before-fix version of its source code package for every bug report. Furthermore, instead of using the package name and the class name, we use the absolute path to indicate the fixed files. We created fine-grained benchmark datasets for evaluation from six open-source projects:

1) Eclipse Platform UI[8]: the user interface of an integrated development platform.
2) JDT[9]: a suite of Java development tools for Eclipse.
3) Birt[10]: an Eclipse-based business intelligence and reporting tool.
4) SWT[11]: a widget toolkit for Java.
5) Tomcat[12]: a web application server and servlet container.
6) AspectJ[13]: an aspect-oriented programming extension for Java.

All these projects use Bugzilla as their issue tracking system and GIT as a version control system (earlier versions are transferred from CVS/SVN to GIT). The bug reports, source code repositories, and API specifications are all publicly accessible.

Bug reports with status marked as *resolved fixed*, *verified fixed*, or *closed fixed* were collected for evaluation. To map a bug report with its fixed files, we apply the heuristics proposed by Dallmeier and Zimmermann in [17]. Thus, we searched through the project change logs for special phrases such as "bug 319463" or "fix for 319463". If a bug report links to multiple git commits or revisions, or if it shares the same commit with others, it will be ignored because it is not clear which fixed file is relevant. Bug reports without fixed files are also ignored because they are considered not functional [17]. Overall, we collected more than 22,000 bug reports from the six projects.

The exact versions of the software packages for which bugs were reported were not all available. Therefore, for each bug report, the version of the corresponding software package right before the fix was committed was used in the experiment. This may not be the exact same version based on which the bug was reported originally. Therefore, the association may not capture exactly what took place in the real world. However, since the corresponding fix had not been checked in, and the bug still existed in that version, it is reasonable to use this association in our evaluation.

For each project and the corresponding dataset, Table 2 shows the time range for the bug reports and a number of basic statistics such as the number of bug reports that were mapped to fixed files, the number of fixed files per bug report, the project size, and the number of API entries (classes or interfaces) from the project API specification that are used during evaluation. The fine-grained benchmark datasets are made publicly available[14].

5 Evaluation Metrics

Given a test dataset of bug reports, testing the model means computing a ranking of the source code files for each bug report in the dataset. For any given bug report r, the system ranking is created by computing the weighted scoring function $f(r,s)$ for each source code file s in the project, followed by ranking all the files in descending order of their scores. The system ranking is then compared with the ideal ranking in which the relevant files are listed at the top. The overall system performance is then computed using the following evaluation metrics:

- *Accuracy@k* measures the percentage of bug reports for which the system makes at least one correct recommendation in the top k ranked files.
- *Mean Average Precision (MAP)* is a standard metric widely used in information retrieval [41]. It is defined as the mean of the Average Precision (AvgP) values obtained for all the evaluation queries:

$$MAP = \sum_{q=1}^{|Q|} \frac{AvgP(q)}{|Q|}, \; AvgP = \sum_{k \in K} \frac{Prec@k}{|K|} \quad (25)$$

Here Q is the set of all queries (i.e., bug reports), K is the set of the positions of the relevant documents in the ranked list, as computed by the system. $Prec@k$ is the retrieval precision over the top k documents in the ranked list:

$$Prec@k = \frac{\text{\# of relevant docs in top } k}{k} \quad (26)$$

- *Mean Reciprocal Rank (MRR)* [66] is based on the position $first(q)$ of the first relevant document in

[8] http://projects.eclipse.org/projects/eclipse.platform.ui
[9] http://www.eclipse.org/jdt/
[10] https://www.eclipse.org/birt/
[11] http://www.eclipse.org/swt/
[12] http://tomcat.apache.org
[13] http://eclipse.org/aspectj/
[14] http://dx.doi.org/10.6084/m9.figshare.951967

TABLE 2
Benchmark Datasets: *Eclipse* refers to Eclipse Platform UI.*

Project	Time Range	# of bug reports mapped	# of fixed files per bug report			# of Java files in different versions of the project source package			# of API entries
			max	median	min	max	median	min	
Eclipse*	2001-10-10 – 2014-01-17	6,495	587	2	1	6,243	3,454	382	1,314
JDT	2001-10-10 – 2014-01-14	6,274	118	2	1	10,544	8,184	2,294	1,329
Birt	2005-06-14 – 2013-12-19	4,178	230	1	1	9,697	6,841	1,700	957
SWT	2002-02-19 – 2014-01-17	4,151	430	3	1	2,795	2,056	1,037	161
Tomcat	2002-07-06 – 2014-01-18	1,056	94	1	1	2,042	1,552	924	389
AspectJ	2002-03-13 – 2014-01-10	593	87	2	1	6,879	4,439	2,076	54

the ranked list, for each query q:

$$MRR = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\frac{1}{first(q)} \tag{27}$$

6 Learning to Rank & Hyperparameters

As introduced in Equation 1 (repeated below for convenience), the ranking model $f(r,s)$ is based on a weighted combination of features that capture domain dependent relationships between a bug report r and a source code file s:

$$f(r,s) = \mathbf{w}^T\Phi(r,s) = \sum_{i=1}^{k} w_i * \phi_i(r,s) \tag{28}$$

The model parameters $\mathbf{w}$ are trained using the learning-to-rank approach from [28], as implemented in the SVM^{rank} package [29]. In this approach, learning $\mathbf{w}$ is equivalent with solving the optimization problem shown in Figure 11 below, in which $\mathcal{R}$ denotes the set of bug reports in a training set, $\mathcal{P}(r)$ is the set of relevant (positive) source files for bug report r, and $\mathcal{N}(r)$ is the set of files that are irrelevant (negative) for bug report r.

minimize:

$$J(\mathbf{w},\xi) = \tfrac{1}{2}\|\mathbf{w}\|^2 + C\sum \xi_{rpn}$$

subject to:

$$\begin{aligned} \mathbf{w}^T\Phi(r,p) - \mathbf{w}^T\Phi(r,n) &\geq 1 - \xi_{rpn} \\ \xi_{rpn} &\geq 0 \\ \forall r \in \mathcal{R}, p \in \mathcal{P}(r), n \in \mathcal{N}(r) \end{aligned}$$

Fig. 11. SVM ranking optimization problem.

The aim of this formulation is to find a weight vector $\mathbf{w}$ such that the corresponding scoring function ranks the files that are known to be relevant for a bug report at the top of the list for that bug report. Thus, if p is known to be relevant (positive) for bug report r and n is known to be irrelevant (negative) for the same bug report, then the objective of the optimization procedure is to find parameters $\mathbf{w}$ such that 1) the number of ranking constraints $\mathbf{w}^T\Phi(r,p) > \mathbf{w}^T\Phi(r,n)$ i.e., $f(r,p) > f(r,n)$, that are violated in the training data is minimized; and 2) the ranking function $f(r,s)$ generalizes well beyond the training data. Furthermore, it can be shown that the learned $\mathbf{w}$ is a linear combination of the feature vectors $\Phi(r,p)$ and $\Phi(r,n)$, which makes it possible to use kernels.

In order to fully specify the training procedure, we need to assign values to a number of hyperparameters. First, we need to determine how many ranking triplets $\langle p \succ n | r\rangle$ should be included in the training data such that training the ranking model is feasible in terms of time complexity while at the same time the resulting model obtains good performance on test data. If possible, the size of the training data should be project independent. The total number N of possible training triplets is indirectly determined by the number of bug reports chosen to be included in $\mathcal{R}$, the number of relevant files in $\mathcal{P}(r)$ and the number of irrelevant files in $\mathcal{N}(r)$, for each bug report $r \in \mathcal{R}$, as shown below:

$$\begin{aligned} N &= \sum_{r\in\mathcal{R}} |\mathcal{P}(r)| \times |\mathcal{N}(r)| \\ &\leq \sum_{r\in\mathcal{R}} |\mathcal{P}(r)| \times (|\mathcal{T}(r)| - |\mathcal{P}(r)|) \end{aligned} \tag{29}$$

The number of relevant files in $\mathcal{P}(r)$ is usually very small – for the six projects considered, the median of this number is between 1 and 3, as shown in Table 2. Consequently, the size of the training dataset will be dominated by the number of bug reports included in $\mathcal{R}$ and the number of irrelevant files used for each bug report. The maximum number of irrelevant source code files that can be used in $\mathcal{N}(r)$ is given by $|\mathcal{T}(r)| - |\mathcal{P}(r)|$ i.e., the difference between the total number of files in the project when the bug report r was submitted and the number of relevant files in $\mathcal{P}(r)$. Since the median number of files in $\mathcal{T}(r)$ is between 1,552 and 4,439 across the six projects, as shown in Table 2, the total number of irrelevant source code files will also be very large. Consequently, including all the irrelevant files in $\mathcal{N}(r)$ for each bug report would result in a very large number N of training triples, which in turn would make training unfeasible in terms of overall time complexity. In Section 6.1 we show that the system performance remains in the optimal range when $\mathcal{R}$ is limited to contain only 500 bug reports and when $\mathcal{N}(r)$ is created

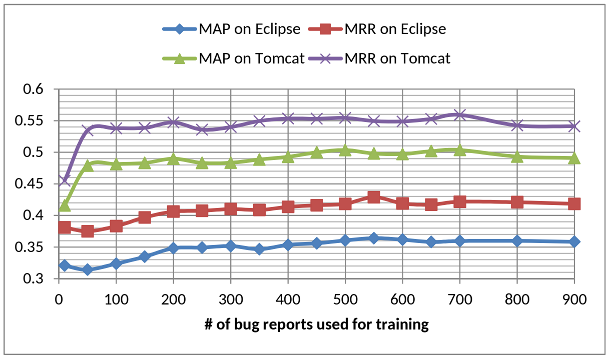


Fig. 12. MAP and MRR as a function of $|\mathcal{R}|$.

from only the 200 irrelevant files that are closest to the bug report r in terms of cosine similarity.

Second, we also need to determine a value for the capacity parameter C that would lead to good performance on the test data. In Section 6.2 we show that the system performance is close to optimal as long as $C \in [10, 100]$.

6.1 Tuning the Size of the Training Dataset

Since utilizing all the irrelevant files for a bug report makes training intractable, we would like to select only a limited subset of irrelevant files to use during training. To maximize their utility, we want to select irrelevant files that are *similar* to the bug report. Therefore, we first use the VSM cosine similarity feature $\phi_1(r, s)$ to rank all the files in the project and then select only the top $\mathcal{N}(r)$ irrelevant files for training. In all experiments, the set of relevant files $\mathcal{P}(r)$ will contain all the source code files that are relevant to the bug report r.

To estimate the optimal number of bug reports $|\mathcal{R}|$ and the optimal number of irrelevant files $|\mathcal{N}(r)| \ll |\mathcal{T}(r) - \mathcal{N}(r)|$, we compute learning curves for two different projects: Eclipse Platform UI and Apache Tomcat. For the Eclipse Platform UI project, we use the latest 500 bug reports for testing, the previous 250 for validation, and the remaining 5,745 as the pool of training examples. Similarly, for the smaller Apache Tomcat project, we use the latest 100 bug reports for testing, the previous 50 for validation, and the remaining 906 as the pool of training examples. To create the learning curves, we repeatedly train the ranking model on $|\mathcal{R}|$ bug reports, where $\mathcal{R}$ is first set to contain the newest 10 bug reports from the training pool, then the newest 50 bug reports, the newest 100 bug reports, up to all the bug reports in the training pool, in increments of 50. For each training sample $\mathcal{R}$, the remaining hyperparameters $|\mathcal{N}(r)|$ and C are tuned using grid search to maximize MAP on the validation data. The resulting learning curves for $|\mathcal{R}|$ are shown in Figures 12. Both MAP and MRR improve substantially as the number of training bug reports in $\mathcal{R}$ increases from 10 to 500. Beyond 500 bug reports, MAP and MRR performance stays mostly flat. Since this behavior is similar for the two different projects, in the remaining experiments we choose training sets $|\mathcal{R}| = 500$ for all six projects.

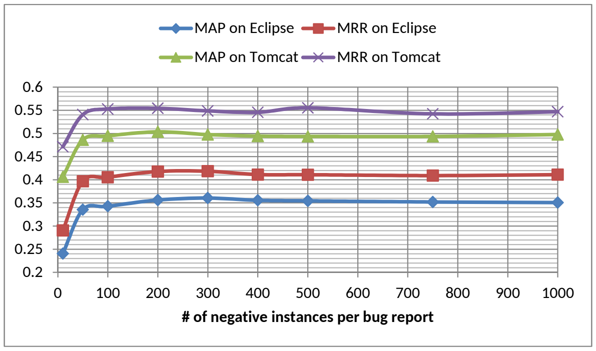


Fig. 13. MAP & MRR as a function of $|\mathcal{N}(r)|$.

To estimate the number of irrelevant files to be included in each $\mathcal{N}(r)$, we use a similar procedure. First, we fix the set of bug reports $\mathcal{R}$ to contain the latest 500 bug reports from the training pool. Then, to create the learning curves, we repeatedly train the ranking model on the bug reports in $\mathcal{R}$, where for each bug report $r \in \mathcal{R}$ we use the same number of irrelevant files in $\mathcal{N}(r)$. The set $\mathcal{N}(r)$ is first set to the 10 irrelevant files that are most similar with the bug report r, in terms of the cosine similarity feature $\phi_1(r, s)$. Then the size of $\mathcal{N}(r)$ is increased to the top 50, 100, up to the top 1000 irrelevant files. For each of the resulting dataset of training triples, the capacity hyperparameter C is tuned using grid search to maximize MAP on the validation data.

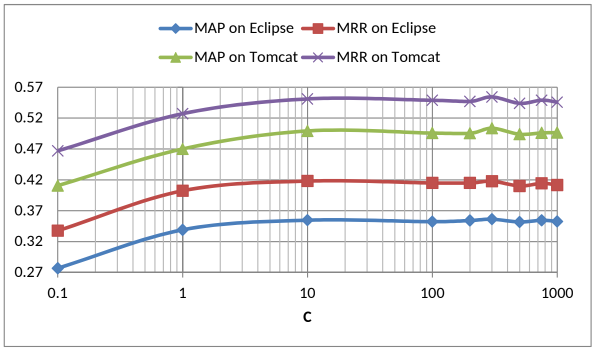


Fig. 14. MAP and MRR as a function of C.

The resulting learning curves for $|\mathcal{N}(r)|$ are shown in Figures 13. Both MAP and MRR improve as the number of irrelevant files in $\mathcal{N}(r)$ grows from 10 to 200. Beyond 200, MAP and MRR performance does not improve anymore. Since the same behavior is observed for both projects, in the remaining experiments we choose to use the top 200 irrelevant files for each bug report.

6.2 Tuning the Capacity Parameter

Based on the same training / testing / validation split from the previous section, the training set $\mathcal{R}$ is set to contain the latest 500 bug reports from the training pool, and for each bug report $r \in \mathcal{R}$ the top 200 irrelevant files are included in $\mathcal{N}(r)$. In order to assess the impact of the capacity parameter on the system performance, we train ranking models with values of C ranging from 0.1 to 1,000. For each value, the trained model is evaluated on the same test dataset.

The system performance as a function of the C parameter is shown in Figures 14. Both projects exhibit a similar behavior: as C increases from 0.1 to 10, both MAP and MRR increase substantially; as C increases from 100 to 1,000, both MAP and MRR are stable, without significant changes. Since the influence of the C parameter appears to be project independent, with a maximum performance at 300, we set $C = 300$ in all the training experiments.

Figures 12 to 14 show similar trends for MAP and MRR. This behavior is expected since the two measures are by definition correlated, e.g. a better ranking that substantially improves the MAP (as happens in the first part of the graphs) is also expected to improve the MRR. MAP and MRR are actually identical for bug reports that have only one relevant source file, which account for over 60% in Eclipse. Furthermore, the two measures quickly reach a plateau and then stay mostly flat. This happens because the same system, when trained with different values of the parameter, outputs rankings that are very similar. We used Spearman's rank correlation to determine the similarity between the source file rankings corresponding to consecutive points in Figure 13, averaged over all the bug reports used for Eclipse, and obtained the following values: {0.799, 0.986, 0.947, 0.980, 0.950, 0.994, 0.996, 0.998}. With the exception of the first two points, where the correlation is about 80%, all the remaining rankings have over 95% correlation. Very similar rankings, by definition, will lead to mostly flat values for both MAP and MRR.

7 Experimental Evaluation

In order to create disjoint training and test datasets, the bug reports from each benchmark dataset are sorted chronologically by their report timestamp. For all the projects but AspectJ, the sorted bug reports are then split into K equally sized folds $fold_1$, $fold_2$, ..., $fold_K$. Thus, $fold_1$ contains the oldest bug reports whereas $fold_K$ contains the latest bug reports. Based on the results of the tuning experiments from Section 6.1, each fold is set to contain 500 bug reports. Correspondingly, the number of folds K for each project is computed as the ratio between the total number of bug reports in the project and the determined optimal size of 500:

$$K = \frac{\text{\# of bug reports}}{500} \tag{30}$$

Thus, the larger projects Birt, Eclipse Platform UI, JDT, SWT, and Tomcat are split into 8, 12, 12, 8, and 2 folds, respectively. The bug reports from AspectJ are split only in 2 folds, due to the smaller size of the project: the oldest 500 for training and the latest 93 for testing.

The ranking model is trained on $fold_k$ and tested on $fold_{k+1}$, for all folds $1 \leq k < K$. Since the folds are arranged chronologically, this means that the system is always trained on the most recent bug reports with respect to the testing fold. The more recent bug reports are expected to better match the properties of the bugs in the current testing fold, which in turn is expected to lead to optimal weights in the ranking function.

For each project, the rankings from all $K - 1$ testing folds are pooled together and the final performance is computed on the pooled rankings using the evaluation measures described in Section 5.

We compared our learning-to-rank (LR) approach with the following 2 baselines:

1) The standard VSM method that ranks source files based on their textual similarity with the bug report.
2) The Usual Suspects method that recommends only the top k most frequently fixed files [30].

We also compared against 3 recent state-of-the-art systems:

3) BugLocator [71] ranks source files based on their size, their textual similarity with the bug report, and information about previous bug fixes.
4) BLUiR [62] computes the textual similarity of the summary and description fields of a bug report with the class name, method names, variable names, and comments within a source code file. The resulting 8 similarity scores are then summed up into an overall ranking score.
5) BugScout [52] classifies source files as relevant or not using an extension of Latent Dirichlet Allocation (LDA) [10].

We re-implemented BugLocator and evaluated it on the fine-grained benchmark dataset, along with our LR approach and the two baselines, as described in Section 7.1 below. Furthermore, since the authors of BugLocator and BLUiR used a different dataset for evaluation, we also evaluated our LR approach on their fixed code revision dataset, as described in Section 7.2 below (but note the problems associated with this dataset, as discussed in Section 4). Lastly, since we were unable to get access to either the tool or the dataset used in [52], in Section 7.3 we compare our system results against the published BLUiR results by testing the LR approach on our replicate of the original dataset used in [52] and [17].

7.1 Comparison with VSM, Usual Suspects, and BugLocator on the Fine-grained Benchmark Dataset

We implemented the two baselines as well as the BugLocator method. We tuned the parameter α of BugLocator on the training data for each project using a grid search from 0.0 to 1.0, with a step of 0.1. We used the tuned α value for testing because we observed it gives better results than the value published in [71].

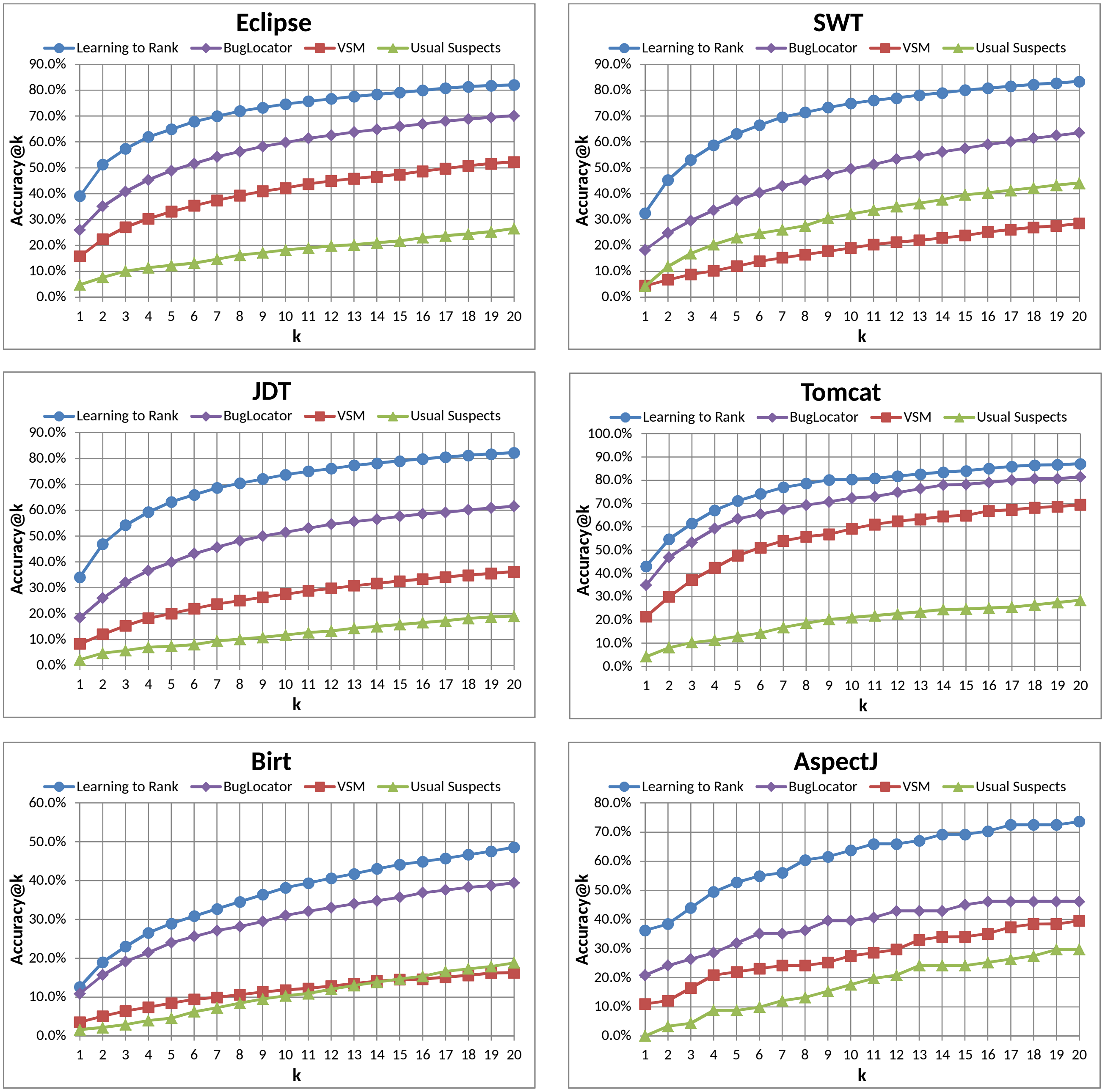
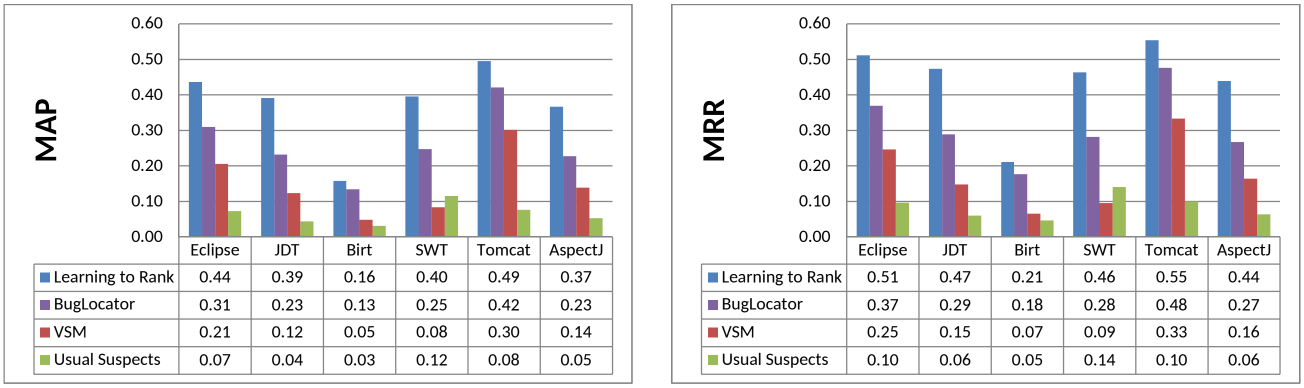


Fig. 15. Accuracy@k graphs of the 4 implemented methods on the 6 benchmark projects.



MAP	Eclipse	JDT	Birt	SWT	Tomcat	AspectJ
🟦 Learning to Rank	0.44	0.39	0.16	0.40	0.49	0.37
🟪 BugLocator	0.31	0.23	0.13	0.25	0.42	0.23
🟥 VSM	0.21	0.12	0.05	0.08	0.30	0.14
🟩 Usual Suspects	0.07	0.04	0.03	0.12	0.08	0.05

MRR	Eclipse	JDT	Birt	SWT	Tomcat	AspectJ
🟦 Learning to Rank	0.51	0.47	0.21	0.46	0.55	0.44
🟪 BugLocator	0.37	0.29	0.18	0.28	0.48	0.27
🟥 VSM	0.25	0.15	0.07	0.09	0.33	0.16
🟩 Usual Suspects	0.10	0.06	0.05	0.14	0.10	0.06

Fig. 16. MAP and MRR comparison of the 4 implemented methods on the 6 benchmark projects.

TABLE 3
Comparison of Learning-to-Rank (LR), BLUiR, and BugLocator (BL) on the same dataset from [62], [71].

Project	# of reports	Metric	LR	BLUiR	BL
Eclipse3.1	3,075	Accuracy@1	**34%**	33%	29%
		Accuracy@5	**57%**	56%	54%
		Accuracy@10	**66%**	65%	63%
		MAP	**0.34**	0.33	0.30
		MRR	**0.45**	0.44	0.41
AspectJ	286	Accuracy@1	32%	**34%**	31%
		Accuracy@5	**56%**	52%	51%
		Accuracy@10	**68%**	62%	59%
		MAP	**0.27**	0.25	0.22
		MRR	0.42	**0.43**	0.41
SWT3.1	98	Accuracy@1	**62%**	56%	40%
		Accuracy@5	**84%**	77%	67%
		Accuracy@10	**89%**	88%	82%
		MAP	**0.62**	0.58	0.45
		MRR	**0.73**	0.66	0.53

The graphs in Figure 15 present the Accuracy@k results for the 4 implemented methods on the 6 benchmark projects, with k ranging from 1 to 20. The LR approach achieves better results than the other three methods on all projects. For example, on the Eclipse Platform UI project our approach achieved Accuracy@k of 39.1%, 57.3%, 64.9%, and 74.6% for k = 1, 3, 5, and 10, respectively. That is to say, if we recommended only one source file to users (k = 1), we would make correct recommendations for 39.1% of the 6,495 collected bug reports. Similarly, if we recommended ten source files (k = 10), we would make correct recommendations for 74.6% of the bug reports. In comparison, BugLocator achieved Accuracy@1 of 25.9% and Accuracy@10 of 58.2%. VSM and Usual Suspects achieved Accuracy@10 of 40.9% and 17.2%, respectively. An application of the Mann-Whitney U Test [40] shows that the LR approach significantly ($p < 0.05$) outperforms BugLocator, VSM and Usual Suspects in terms of Accuracy@k for all six projects, at all values of k.

Tomcat is a project where BugLocator performs closest to the LR approach. Numerous bug reports in Tomcat contain rich descriptions that share many terms with the relevant files, which could explain the relatively high performance obtained by VSM. Since both LR and BugLocator exploit textual similarity between bug reports and source files, it is therefore expected that they perform comparably on this dataset.

Figure 16 compares the 4 methods in terms of MAP and MRR. Here too, the LR approach outperforms the two baselines and BugScout on all 6 projects. For example, on the Eclipse Platform UI project, the MAP and MRR results for the LR approach are 0.44 and 0.51, which compare favorably with Bug Locator (0.31 and 0.37), VSM (0.21 and 0.25), and Usual Suspects (0.07 and 0.10).

7.2 Comparison with BugLocator and BLUiR on the Fixed Code Revision Dataset

The dataset used in [62], [71] was created from a fixed revision of 3 software projects: Eclipse 3.1, AspectJ, and SWT. We split the 3,075 bug reports from Eclipse 3.1 into 6 folds as follows: $fold_6$ contains the newest 575 reports, while each of the 5 remaining folds contains 500 reports. We then train the ranking model on $fold_k$ and test it on $fold_{k+1}$, for all $k \in [1, 5]$. To obtain test results on $fold_1$ too, we train the model on $fold_2$ and test on $fold_1$. We reuse the ranking model that was trained on $fold_2$ of Eclipse 3.1 to test on the data from SWT 3.1 and AspectJ, since these two later projects do not contain enough bug reports for training. Although the LR model was trained on a different project, for AspectJ and SWT it still obtains results that are in general superior to BugLocator and BLUiR, as shown in Table 3. When trained and tested on the same project (Eclipse 3.1), the model is superior to BugLocator and BLUiR on all the performance measures.

The comparison between the LR approach and BugLocator from Section 7.1 is further confirmed in Table 3: the LR approach obtains better results on both the fine-grained benchmark dataset and the fixed code version dataset from [62], [71].

BLUiR syntactically parses a source code file and extracts four types of fields: class names, methods, variables, and comments. The textual similarity between the bug report (itself segmented into a description and a summary field) and the source code file is then computed separately for each field type. The resulting similarity values are summed up into an overall similarity measure. In comparison, the LR approach uses a richer set of similarity functions that it combines into the final ranking score using a weighted sum, where the weights are trained automatically using a learning-to-rank technique. Furthermore, because it relies on textual similarity, BLUiR cannot work well on bug reports that do not have many tokens in common with the relevant source code file. Our approach alleviates this problem by using project API documents to bridge the lexical gap between bug reports and source code files.

As demonstrated in Section 4, using a fixed version of the source code package may introduce future bug-fixing information in the evaluation. While the evaluation results obtained on such a dataset can be used to compare different approaches to file ranking for bug reports, the actual performance numbers shown in Table 3 are likely to overestimate the true performance. In order to get a more realistic estimate of the true performance of these systems, we recommend using the fine-grained benchmark dataset from Section 4, for which the evaluation results are shown in Section 7.1.

7.3 Comparison with BugScout on a Replicated Dataset

Compared to BugLocator, BugScout is a more complex approach and thus more difficult to implement correctly.

Since we were unable to get access to either the tool or the actual dataset used in [52] , we tried to recreate their dataset by following the description from [17], [52]. Thus, we created a test dataset by collecting the specified number of fixed bug reports from the Eclipse Platform (4,136) and AspectJ (271) projects, going backwards from the end of 2010 (when BugScout was published). To train our LR method, we used bug reports that were solved before the bug reports in the testing dataset.

Table 4 shows the Accuracy@k results of the BugScout (BS) and the LR method, for k = 1, 10, and 20. The BugScout results are copied from [52] and are substantially lower than the LR results. While it is possible that our replicated dataset is not entirely identical with their dataset, we believe there are two main reasons why the LR method obtains better results:

1) LR uses features that capture domain knowledge relationships between bug reports and source files.
2) LR is trained to optimize ranking performance, whereas BugScout is trained for classification into multiple topics. Training for classification may represent a mismatch with the testing procedure, when the system is evaluated for ranking.

TABLE 4
Comparison between BugScout (BS) and Learning-to-Rank (LR) on data replicated from [52].

Project	Accuracy@1		Accuracy@10		Accuracy@20	
	BS	LR	BS	LR	BS	LR
Eclipse	<15%	39%	<35%	74%	<40%	82%
AspectJ	<15%	36%	<40%	64%	<60%	74%

Various IR approaches have been developed for the task of identifying source files that are relevant for an input bug report. These include approaches based on LDA [39], [52], [61], Latent Semantic Indexing (LSI) [61], [71], Smoothed Unigram Model (SUM) [61], [71], and SVMs [30], [52]. Since BugLocator was reported to outperform the existing approaches using LDA, LSI, and SUM [71], and since BugScout was reported to outperform the SVM model proposed in [52], we expect our LR system to compare favorably with all these previous approaches.

7.4 Fine-Grained Ranking

Research in the more general area of feature location (discussed in Section 10.2) has led to methods for ranking smaller functional units (e.g. methods), with respect to user queries (e.g. bug reports). Although the main purpose of our work is a method for mapping bug reports to relevant source code files, in this section we describe and evaluate a simple adaptation of our learning-to-rank system that enables it to perform a finer-grained ranking, at the level of methods. Given a bug report r, a method-level ranking is computed as follows:

1) Rank all source code files s based on their scores $f(r, s)$, as computed by the LR system using all 19 features (Equation 1).
2) Within each source file, rank all its methods m based on their lexical similarities with the bug report $sim(r, m)$ (Equation 4).
3) Eliminate from the ranking all methods m for which $sim(r, m) < \tau$ i.e. their lexical similarity with the bug report is below a pre-defined threshold τ.

Equivalently, if m_1 and m_2 are two methods from source code files s_1 and s_2, respectively, such that both $sim(r, m_1) \geq \tau$ and $sim(r, m_2) \geq \tau$, then m_1 is ranked higher than m_2 if and only if:

- $s_1 \neq s_2$ and $f(r, s_1) > f(r, s_2)$.
- $s_1 = s_2$ and $sim(r, m_1) > sim(r, m_2)$.

We compare this simple adaptation of our system with the state-of-the-art feature location approach introduced in [22]. In that approach, the lexical similarity between a method and a query is measured using the LSI technique from IR. The PageRank and HITS algorithms are used on the method-call graph to filter out methods that share less connections with others. Irrelevant methods are further filtered out based on the runtime execution trace. Several of our proposed features are similar with their features, although they work on a different level of granularity (file vs. method). Also, a data fusion technique is used in [22] to combine the different features, whereas we use a learning-to-rank SVM.

For comparison purposes, we evaluate our system on the same 45 bug reports from Eclipse that were used in [22][15] and report performance in terms of *effectiveness*, which is defined as the position of the top ranked relevant method that needs to be fixed for the input bug report [22], [56]. To train the file ranking function $f(s, r)$, we used the Eclipse bug reports that were fixed before the 45 bugs were reported.

7.4.1 *Results and Analysis*

The first step in our adapted system is to rank all the source code files for every bug report in the dataset. Table 5 shows the file-level ranking results in terms of Accuracy, MAP, and MRR. The ranking performance on these 45 Eclipse bug reports is lower than the performance shown in Section 7.1, which was obtained on the 6,495 Eclipse bug reports from our fine-grained benchmark dataset. The main reason for this difference is that the 45 bug reports are from 2004 and therefore there is not much historical information that can be used for computing features that are based on collaborative filtering or the file revision history. In particular, there is less opportunity for exploiting duplicated bug reports.

TABLE 5
The results of ranking 12,359 source files for the 45 Eclipse 3.0 bug reports used in [22].

Accuracy@1	Accuracy@5	Accuracy@10	MAP	MRR
29%	33%	40%	0.29	0.32

[15] http://www.cs.wm.edu/semeru/data/emse-link-analysis/

After ranking the source code files, the next two steps in the algorithm rank the methods within each file and filter out methods whose similarity with the bug reports falls below a predefined threshold $\tau = 0.02$, whose value was tuned on a separate development set. Table 6 shows the percentage of bug reports for which at least one relevant method can be located by the adapted system in the top N recommended methods, for two values of the similarity threshold: $\tau = 0$ i.e. no method is filtered out, and the tuned value $\tau = 0.02$. The results show the utility of filtering out low scoring methods: when recommending the top 250 methods, the thresholded system is able to locate a relevant method for 49% of bug reports, whereas the un-thresholded system needs to recommend almost twice the number of methods to achieve the same percentage.

TABLE 6
The percentage of bug reports for which at least one relevant method can be located, when recommending the top N out of 131,802 methods in Eclipse 3.0.

	29%	49%	60%	80%	84%	87%
Top N ($\tau = 0$)	20	450	1,000	3,000	5,000	7,000
Top N ($\tau = 0.02$)	20	250	350	1,500	1,600	3,700

Table 7 shows a comparison between the *effectiveness* our adapted system for method ranking (MR) and the state-of-the-art method $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}^{bot}$ introduced in [22] (henceforth abbreviated as IRDynWM). The left most column shows the percentage of bug reports for which at least one relevant method (for the last three columns) or file (for the second column) can be located by each system. Different percentages are obtained by varying the number N of top recommendations returned by the system. Each cell contains the corresponding *median* / *mean* effectiveness, with the better numbers indicated in bold.

The adapted system MR ($\tau = 0.02$) achieved a better performance in terms of *effectiveness* when it was tuned to locate a relevant method for up to 60% of bug reports. For example, for 29% of bug reports, it can locate at least one relevant method with only 8 recommendations on average, while IRDynWM needs to make 68 recommendations. On the other hand, IRDynWM achieves better performance when tuned to locate a relevant method for 80% of bug reports or more. Thus, in order to locate a relevant method for 87% of bug reports, our system MR ($\tau = 0.02$) needs to make an average of 502 recommendations, while IRDynWM needs to recommend only 425 methods. Equivalently, when making up to 100 recommendations, our system can locate at least one relevant method for more bug reports than IRDynWM. But when recommending more than 300 methods, IRDynWM performs better.

TABLE 7
Comparison between the adapted Learning-to-Rank (LR) approach for method ranking (MR) and the best performing feature location method of Dit et al. [22].

	file-level	*method-level*		
	LR	MR ($\tau = 0$)	MR ($\tau = 0.02$)	IRDynWM [22]
29%	1 / 1	6 / 8	**6 / 8**	41 / 68
49%	1 / 4	18 / 93	**18 / 67**	56 / 116
60%	3 / 10	40 / 233	**34 / 120**	76 / 202
80%	10 / 28	278 / 661	169 / 321	**87 / 219**
84%	11 / 31	295 / 782	217 / 358	**111 / 319**
87%	13 / 47	361 / 1060	244 / 502	**152 / 425**

We noticed that all 45 bug reports used in the evaluation dataset describe the process of reproducing the bug in details. Dit et al. [22] use this detailed information to reproduce the bug and obtain runtime execution traces. Runtime execution traces are also used in other method-level feature location systems: Liu et al. [38] use LSI to rank only methods in the execution trace; Antoniol et al. [2] collect method-level execution traces and rank methods based on the frequencies of method calls; Poshyvanyk et al. [56] combine an LSI-based ranking with a scenario-based probabilistic ranking of method and function calls observed while executing a program. Our model does not use execution traces and therefore does not need to reproduce the bug. On the other hand, we use features derived from the file revision history, a source of evidence that is not used by IRDynWM. The performance differences shown above can be explained by the differences in the feature sets, coupled with the use of different training procedures. Given the complementary behavior of the two systems, a worthwhile direction for future work is to combine their features in a more comprehensive approach.

8 Feature Analysis

The results reported in Section 7 above correspond to using all 19 proposed features. However, some features may become redundant when used in conjunction with other features, or they may be entirely irrelevant to the task. While all the proposed features were justified by our theoretical knowledge of the task properties, in this section we seek to give an empirical justification of their utility for the ranking task. More exactly, we employ feature selection techniques in order to answer the following research questions:

1) Is there a strict subset of features that obtains a better performance than using all the features?
2) Are there any irrelevant features?
3) What are the features that have the most impact on the ranking performance?

To answer these questions we use the greedy backward elimination algorithm for feature selection [60], [67]. Although filter methods are known to obtain good performance for defect prediction [64], our use of greedy backward elimination is motivated by research showing that wrapper methods are better at producing features sets that fit the learning algorithm [34], albeit in a more computationally intensive manner.

```
Algorithm 1 FEATURESELECTION($F$, $\mathcal{L}$, $D_{train}$, $D_{tune}$)
Input: A set of features $F$ and a learning algorithm $\mathcal{L}$.
       Training and tuning data $D_{train}$ and $D_{tune}$.
Output: A selected set of features $S \subseteq F$.
MAP$_{global} \leftarrow 0$
while $|F| > 1$ do
    MAP$_{local} \leftarrow 0$
    for all features $\phi \in F$ do
        train $\mathcal{L}$ on $D_{train}$ using features $F - \{\phi\}$.
        if MAP on $D_{tune} >$ MAP$_{local}$ then
            MAP$_{local} \leftarrow$ MAP
            $F' \leftarrow F - \{\phi\}$
    $F \leftarrow F'$
    if MAP$_{local} >$ MAP$_{global}$ then
        MAP$_{global} \leftarrow$ MAP$_{local}$
        $S \leftarrow F'$
return $S$
```

Algorithm 1 shows the pseudocode of the greedy backward feature selection procedure used in our experiments. At each iteration, the algorithm greedily removes a feature from the current set of features such that the MAP on the tuning dataset is maximized. The feature set that obtains the best MAP across all iterations is returned. To estimate the utility of the proposed features, we used the feature selection procedure in two settings:

- [**Feature Selection**] In the first setting, to be described in Section 8.1 below, the features were evaluated on tuning data that was separate from the test data, i.e. $D_{tune} \neq D_{test}$. This means that the test performance using the automatically selected features reflects the expected performance if those features were used in a practical system on unseen bug reports.
- [**Feature Comparison**] In the second setting, to be described in Section 8.2 below, the features were evaluated directly on the test data, i.e. $D_{tune} = D_{test}$. Although the optimal test performance in this setting is likely to overestimate the true performance on unseen bug reports, the results are meant to be used mainly for determining the relative utility of features for the ranking task.

In both settings, we use a k-fold scenario for evaluation in order to obtain test results over as much data as possible. As will be seen in Section 8.1 below, the **Feature Selection** setting selects a slightly different feature set for each fold. Following [19], in the **Feature Comparison** setting we estimate just one globally optimal feature set by pooling the test results across all folds, and greedily sort the features by identifying at each step the feature whose removal leads to the best performance.

8.1 Feature Selection

For a dataset that is split into K folds, we evaluate the LR model with automatically selected features, as follows:

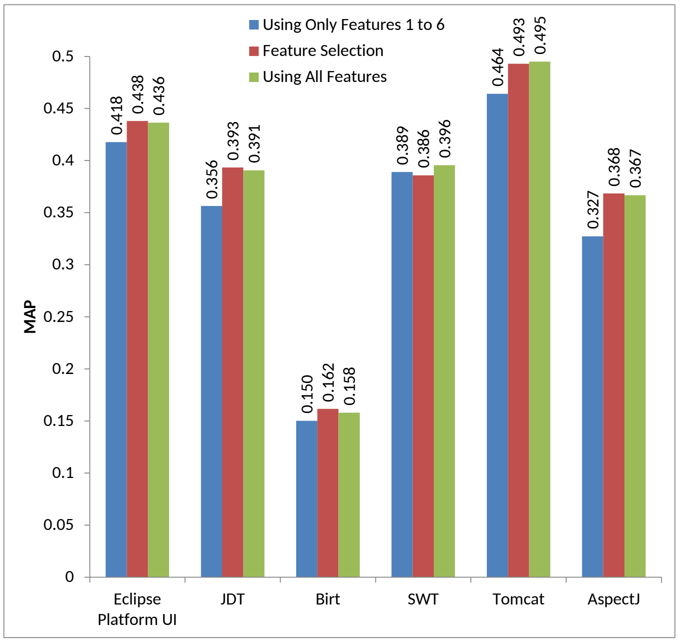


Fig. 17. MAP results for 3 types of feature sets.

1) **Feature Selection**: Run the greedy backward feature elimination Algorithm 1 by setting $D_{train} = fold_k$ and $D_{tune} = fold_{k+1}$.
2) **Evaluation**: Train model with selected features on $fold_{k+1}$ and test it on $fold_{k+2}$.

We run these two steps for all folds $1 \leq k \leq K-2$ and compute the MAP over the results pooled from the $K-2$ testing folds: $fold_3$ to $fold_K$.

We apply the above procedure on Eclipse Platform UI with 12 data folds, JDT with 12 data folds, Birt with 8 data folds, and SWT with 8 data folds. Since AspectJ and Tomcat contain an insufficient number of bug reports for feature selection, we select features based on their shared utility for the other 4 projects (Birt, Eclipse Platform UI, JDT, and SWT) in two steps, as follows. In the first step, using the results of the feature comparison from Section 8.2, we identify for each of the 4 projects the set of features that maximizes the MAP on their test data. These feature sets are shown in the green background bands in Figures 19 and 20. Then, in the second step, we create a feature set for AspectJ and Tomcat by selecting only those features that appear in at least 3 of the 4 feature sets that maximize the MAP. The resulting feature set is $S' = S - \{\phi_9, \phi_{13}, \phi_{16}\}$, where S is the feature set with all 19 features. This feature set is then used to train on $fold_1$ and test on $fold_2$ for both AspectJ and Tomcat.

Figure 17 compares the MAP results obtained for 3 different feature sets, across all 6 benchmark projects. The left blue bar corresponds to using only the six features ϕ_1 to ϕ_6 that were introduced in our previous work [70]. The middle red bar corresponds to the feature sets that were automatically selected using the procedure above. The green red bar shows the MAP result when all 19 features are used. The results show that the model using all 19 features always achieves better performance than the model using only the first 6 features. The relative improvement in terms of MAP for the six projects is 4.3%, 9.8%, 5.3%, 1.8%, 6.7%, and 12.2%, respectively. Applying feature selection further improves the model performance in terms of MAP on Eclipse Platform UI, JDT, and Birt, respectively, but hurts performance on SWT. For the smaller projects AspectJ and Tomcat, the MAP results when using features selected from the other four projects are similar with the results when all features are used. We applied the Mann-Whitney U Test [40] to test whether there is a significant difference between the results of feature selection (red bars) and the results of using all features (green bars). The *p-values* shown in Table 8 are all less than 0.88, which means that MAP differences between using feature selection and using all features are not statistically significant.

TABLE 8
MAP Comparison: Feature Selection vs. All Features.

Project	# of test folds	MAP across all test folds		p value
		Feature Selection	All Features	
Eclipse	10	0.438	0.436	0.880
JDT	10	0.393	0.391	0.821
Birt	8	0.162	0.158	0.749
SWT	8	0.386	0.396	0.873

Therefore, to answer the first research question asked earlier in Section 8, automatic feature selection leads to subsets of features that overall achieve similar performance when compared with using all the features.

8.1.1 Are there any irrelevant features?

Figure 18 shows for each of the 4 larger projects a histogram with the number of test folds for which a feature was selected by the greedy backward elimination algorithm. On the right hand side are the corresponding radar charts, in which features lie on the circumference and their selection occurrence is shown on the radius. In the Eclipse Platform UI benchmark project, for example, there are $K = 12$ folds, out of which $fold_3$ to $fold_{12}$ are used for testing. A feature set is selected by Algorithm 1 for each of the 10 testing folds. Correspondingly, Figure 18 shows that features ϕ_1 and ϕ_3 appear in the feature sets of all 10 folds, whereas feature ϕ_8 is selected only once.

Looking at Figure 18, we see that all 19 features are selected at least once in every project. A feature that is not selected for one fold may be selected for another fold. Furthermore, a feature such as ϕ_6 that is less important in one project, such as Eclipse Platform UI, may be more important in another project, such as SWT. Therefore, to answer the second research question asked earlier in Section 8, all features appear to be relevant for the ranking task.

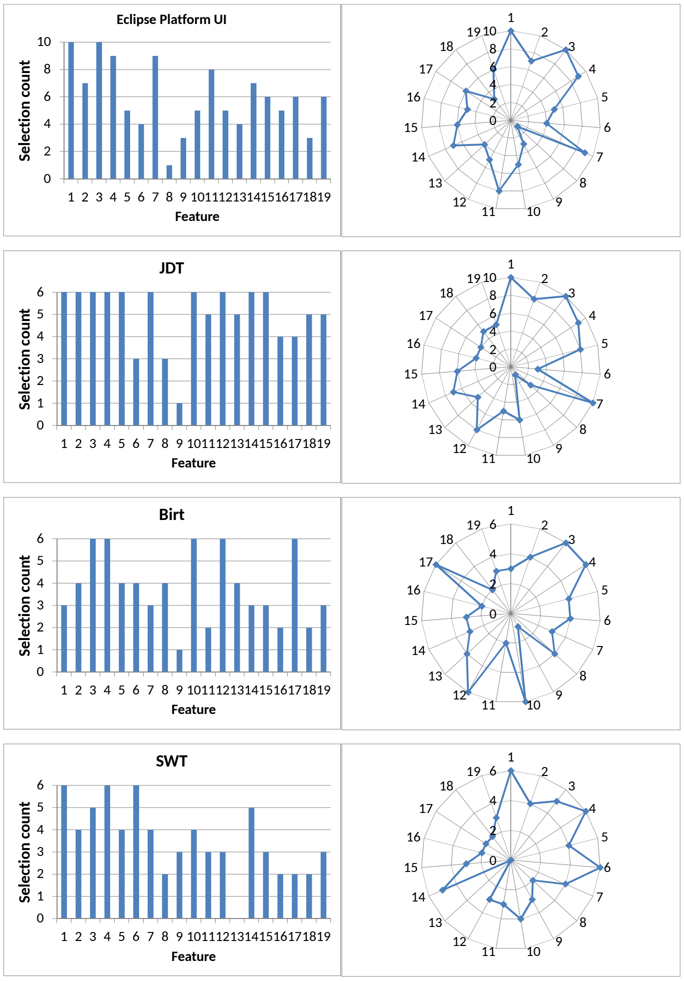


Fig. 18. Feature selection histograms.

8.2 Feature Comparison

To estimate the impact of each feature on the test performance, we run a modified version of the greedy backward elimination algorithm, this time using the test data as tuning data, i.e., $D_{tune} = D_{test}$, as originally proposed on a different problem in [19]. In order to obtain an MAP over all the test folds, we change lines 5 and 6 in Algorithm 1 such that, for a dataset that is split into K folds, lines 5 and 6 repeatedly train and test on $(D_{train}, D_{tune}) = (fold_k, fold_{k+1})$, for all folds $1 \leq k \leq K - 1$. The MAP in line 6 is then computed over the test results pooled over all $K - 1$ test folds. For convenience, the modified procedure is shown in the Feature Comparison Algorithm 2 below.

Lines 6 to 9 in Algorithm 2 determine which feature can be removed from the current set of features such that it maximizes the MAP obtained with the remaining features. Taking Eclipse Platform UI as an example, the MAP's computed in lines 6 to 9 for each feature are shown in the horizontal lines in the triangle from Figure 19. Thus, the first time the loop body in lines 6–9 is executed, the feature whose removal leads to the optimal MAP of 0.4380 is $\phi_{max} = \phi_{16}$, shown in red

Algorithm 2 FEATURECOMPARISON(F, $\mathcal{L}$, D, K)

```
Input: A set of features $F$ and a learning algorithm $\mathcal{L}$.
       A dataset $D$ split into $K$ folds.
Output: A ranked sequence of features $S$.
$S \leftarrow \langle\rangle$
while $|F| > 1$ do
    $\text{MAP}_{max} \leftarrow 0$
    for all features $\phi \in F$ do
        for $k = 1$ to $K - 1$ do
            train $\mathcal{L}$ on $fold_k$ and test on $fold_{k+1}$, using
            features $F - \{\phi\}$.
        if MAP $>$ $\text{MAP}_{max}$ then
            $\text{MAP}_{max} \leftarrow$ MAP
            $\phi_{max} \leftarrow \phi$
    $F \leftarrow F - \{\phi_{max}\}$
    $S \leftarrow \langle\phi_{max}, S\rangle$
$S \leftarrow \langle F, S\rangle$
return $S$
```

on the first line in the triangle. Next time the loop is executed, feature ϕ_9 is removed. The algorithm continues removing features that are increasingly important until the feature set contains only feature ϕ_1.

The header line in Figure 19 shows from right to left all 19 features, in the order in which they are removed by the greedy backward elimination procedure. Correspondingly, we could say that the same header line lists, from left to right, all 19 features in the order of their importance, or impact, for the Eclipse Platform UI benchmark project. Thus, the most important feature for this project appears to be feature ϕ_1. Looking at all the MAP values shown in red, we observe that the globally optimal MAP value is 0.4413, which is obtained for the feature set shown on green background in the header line. Figure 20 shows only the header lines, together with the best MAP at every iteration of Algorithm 2 for all 6 benchmark projects. The numbers in bold indicates the best performance across all iterations, and correspond to the sets of features shown on green background.

The feature ranking results on the test data confirm the results obtained with automatic feature selection in the previous section. For example, Figure 19 shows that the top 4 most important features in the Eclipse Platform UI benchmark project are ϕ_1, ϕ_3, ϕ_4, and ϕ_7. These are also the features that have been selected most often in the automatic feature selection experiments, according to the histogram shown in Figure 18. Similarly, the feature ranking triangle and the feature histogram both show that features ϕ_{16} and ϕ_8 are the least important in both types of experiments.

Overall, the results in Figure 20 show that the six most important features are responsible for more than 90% of the system MAP performance. For example, using the top ranked features $\{1, 7, 4, 3, 15, 11\}$ for Eclipse Platform UI, the system achieves an MAP of 0.4344, which is 98.4% of the best MAP of 0.4413 (shown in bold). On the other hand, when using only the top ranked feature ϕ_1, the learning-to-rank system achieves an MAP of 0.3398, whereas using all the features results in an MAP of 0.4380, which represents a substantial 28.9% relative improvement over the simple VSM feature. These results can be used to inform a trade-off between ranking accuracy (MAP) and system complexity (less features means easier implementation and lighter CPU and memory loads at runtime).

8.2.1 *What features have the most impact?*

According to both the feature histograms in Figure 18 and the feature rankings in Figure 20, feature ϕ_1, which measures the lexical similarity between bug reports and source code, is the most important feature for all the projects but Birt. For Birt, feature ϕ_1 is, surprisingly, the first feature to be removed. Looking at the performance of the LR model on Birt, we notice that it is lower than the performance for the other projects, as shown in Figures 16 and 17. A possible inference here is that if the lexical similarity feature ϕ_1 does not help in mapping a bug report to a relevant source code file, then it is difficult to achieve a good ranking performance, no matter which of the remaining features are added to the model.

Feature ϕ_3, which measures the lexical similarity between a new bug report and the previously fixed bug reports, is another important feature, as it is ranked in the top 5 features in every project. Feature ϕ_4, which measures the class name similarity between bug reports and source files, appears in the top 4 features in all projects, with the exception of Tomcat. The API-enriched lexical similarity feature ϕ_2 helps especially on JDT, Birt, and Tomcat. The feature ϕ_5 that measures the lexical similarity between comments in source code and the bug report helps on all projects. Overall, the most important features for every project are all query-dependent features, which is expected since query-independent features do not take into account the content of the bug report. Of the features that are independent of the content of the bug report, feature ϕ_6 that measures how frequently a source file was fixed before helps especially on SWT, AspectJ, and Tomcat.

The utility of the newly added features (7 to 19) is mostly project dependent. According to Figure 20, feature ϕ_7, which measures the textual similarity between the bug report summary and the class names in a source file, is very useful for Eclipse Platform UI (penultimate to be eliminated) but not so useful for Tomcat and AspectJ (second to be eliminated). Similarly, the query-independent feature ϕ_{15}, which counts the number of file dependencies, is much more important for Eclipse and Tomcat than for AspectJ. Feature ϕ_{17}, the PageRank score on the file dependency graph, is found in the optimal feature set for 3 of the 6 projects evaluated (the green bands in Figure 20). Feature ϕ_{19}, the Hub score in the file dependency graph, is even more useful, as it appears in the optimal feature sets for 5 projects.

Eclipse Platform UI

Feature	1	7	4	3	15	11	10	2	6	19	17	18	13	12	5	14	8	9	16
MAP	0.4091	0.4334	0.4197	0.4186	0.4370	0.4347	0.4353	0.4348	0.4354	0.4360	0.4371	0.4356	0.4371	0.4371	0.4377	0.4362	0.4367	0.4374	0.4380
	0.4076	0.4348	0.4207	0.4201	0.4355	0.4357	0.4350	0.4357	0.4345	0.4353	0.4363	0.4361	0.4370	0.4362	0.4369	0.4354	0.4359	0.4372	
	0.4087	0.4353	0.4206	0.4208	0.4379	0.4343	0.4342	0.4348	0.4354	0.4355	0.4375	0.4362	0.4374	0.4378	0.4366	0.4371	0.4382		
	0.4089	0.4354	0.4227	0.4213	0.4367	0.4352	0.4364	0.4374	0.4359	0.4396	0.4390	0.4363	0.4377	0.4387	0.4373	0.4398			
	0.4067	0.4366	0.4236	0.4202	0.4375	0.4371	0.4337	0.4357	0.4373	0.4369	0.4381	0.4377	0.4400	0.4393	0.4401				
	0.4072	0.4377	0.4197	0.4208	0.4370	0.4347	0.4349	0.4371	0.4374	0.4374	0.4378	0.4380	0.4398	0.4406					
	0.4020	0.4352	0.4190	0.4213	0.4389	0.4350	0.4359	0.4369	0.4380	0.4387	0.4374	0.4386	0.4405						
	0.4022	0.4364	0.4216	0.4216	0.4399	0.4368	0.4362	0.4377	0.4377	0.4377	0.4373	**0.4413**							
	0.4022	0.4357	0.4229	0.4228	0.4359	0.4356	0.4368	0.4367	0.4401	0.4401	0.4402								
	0.4040	0.4368	0.4219	0.4230	0.4356	0.4354	0.4370	0.4383	0.4396	0.4401									
	0.4037	0.4356	0.4220	0.4218	0.4371	0.4346	0.4369	0.4364	0.4383										
	0.3996	0.4239	0.4213	0.4218	0.4362	0.4353	0.4361	0.4378											
	0.3903	0.4219	0.4204	0.4230	0.4320	0.4324	0.4344												
	0.3827	0.4154	0.4178	0.4183	0.4307	0.4309													
	0.3585	0.4028	0.4156	0.4101	0.4280														
	0.3495	0.3995	0.4031	0.4079															
	0.3068	0.3668	0.3741																
	0.1924	0.3398																	

Fig. 19. MAP results on Eclipse Platform UI, as features are removed greedily, from right to left.

Eclipse Platform UI

Feature	1	7	4	3	15	11	10	2	6	19	17	18	13	12	5	14	8	9	16
MAP	N/A	0.3398	0.3741	0.4079	0.4280	0.4309	0.4344	0.4378	0.4383	0.4401	0.4402	**0.4413**	0.4405	0.4406	0.4401	0.4398	0.4382	0.4372	0.4380

JDT

Feature	1	3	7	4	2	12	10	16	5	18	19	14	11	15	8	6	17	13	9
MAP	N/A	0.2433	0.3235	0.3590	0.3818	0.3861	0.3893	0.3923	0.3935	0.3942	0.3977	0.3978	0.3983	0.3992	0.3983	0.3989	**0.3993**	0.3983	0.3961

Birt

Feature	3	4	11	10	2	8	5	13	6	15	12	7	19	17	18	14	9	16	1
MAP	N/A	0.0868	0.1077	0.1378	0.1505	0.1596	0.1628	0.1666	0.1692	0.1705	0.1710	0.1720	0.1725	0.1715	0.1727	0.1727	**0.1735**	0.1718	0.1722

SWT

Feature	4	6	1	3	11	5	10	7	14	12	8	9	13	18	15	17	2	19	16
MAP	N/A	0.2397	0.3168	0.3508	0.3786	0.3868	0.3936	0.3997	0.4006	0.4006	0.4020	0.4015	0.4008	0.4007	0.4003	**0.4030**	0.4022	0.4014	0.3984

Tomcat

Feature	1	3	6	10	2	14	15	19	12	4	16	17	18	8	13	9	11	7	5
MAP	N/A	0.4416	0.4685	0.4905	0.5049	0.5049	0.5093	0.5128	0.5172	0.5191	0.5209	**0.5232**	0.5218	0.5207	0.5219	0.5210	0.5195	0.5195	0.5130

AspectJ

Feature	1	12	4	6	3	10	9	11	18	2	17	19	13	16	14	15	8	7	5
MAP	N/A	0.2968	0.3066	0.3232	0.3224	0.3519	0.3624	0.3666	0.3765	0.3764	0.3751	0.3790	0.3818	**0.3827**	0.3805	0.3802	0.3768	0.3773	0.3766

Fig. 20. MAP results on the 6 projects, as features are removed greedily, from right to left.

To summarize the answer to the third question asked earlier in Section 8, features ϕ_1, ϕ_3, and ϕ_4 are the most important features, overall. Other features, like ϕ_6, ϕ_{10}, ϕ_{15}, and ϕ_{19}, although less important, appear in the optimal feature sets for most projects, which indicates that they provide complementary information that further improves the ranking performance. Finally, the query-dependent features are more important than the query-independent features.

8.3 Summary of Feature Analysis

The feature analysis experiments lead to the following main observations:

1) All 19 features are relevant. If one is interested in a single set of features that has optimal performance across different projects, we recommend using all features. With a sufficient number of training examples (Section 6.1), the model learns project-specific weights that effectively quantify the relative utility of each feature.
2) If one is interested in obtaining optimal performance with a reduced number of features, we recommend automatic feature selection, if the project is sufficiently large. The selected features will be project dependent.
3) For each project, more than 90% of the ranking performance can be obtained using only six features.
4) The lexical similarity between the content of the source code and the content of the bug report is the most important feature.
5) The lexical similarity between the new bug report

TABLE 9
Overall Runtime Performance

Project	Feature Computation (s)		Training (s)		Ranking (s)	
	max	avg.	max	avg.	max	avg.
Eclipse	4940.25	58.13	4.43	4.16	0.02	0.02
JDT	6405.41	279.28	4.31	3.68	0.03	0.03
Birt	5655.69	55.47	5.08	4.50	0.05	0.05
SWT	2831.07	54.91	4.07	3.86	0.02	0.02
Tomcat	754.95	18.11	3.76	3.76	0.02	0.02
AspectJ	1650.83	20.41	4.77	4.77	0.02	0.02

and previous bug reports is another feature that has a significant impact on the ranking performance.

6) Project API documents and file dependency relationships provide complementary information that may further improve the model performance.
7) The utility of most of the newly added features is project dependent.
8) Query-dependent features are more important than query-independent features.

9 Runtime Performance

We performed time-usage and memory-usage evaluations on a computer with CPU Intel(R) Core(TM) i7 920 2.67GHz (8 cores), 24G RAM, and Linux 3.2. Table 9 presents the overall runtime performance of our approach when using all 19 features, across the 6 benchmark datasets. Table 10 shows a breakdown of the time and memory complexity for every feature, as computed in the Eclipse Platform UI project.

9.1 Overall Runtime Performance

The feature computation time, reported in seconds in Table 9, refers to the time needed to calculate all 19 features for all source code files in a project. Given a bug report, the feature computation time includes: the time used to checkout a before-fix version of the source code package, the time used to read the content of every source file on disk, the time used to parse every source file, the time used to index every source file, and finally the time used to calculate the 19 features for all source files.

Although the maximum feature computation time is relatively high, it only happens once for each project. When the first bug report is received, the system indexes all source files for the before-fix version of that bug report. For the vector space model, indexing the source code files requires creating a postings list and a corresponding term vocabulary [41]. To efficiently perform evaluation on over 22,000 subsequent before-fix project versions, the system indexes only the changed files in the subsequent versions. For example, if bug 420972 is the first bug report to be processed in the Eclipse project, the system checks out its before-fix version “2143203” and indexes the corresponding 6,188 source files. When bug 423588 is received later, the system checks out its before-fix version “602d549” and uses the `git diff` command to obtain the list of changed (“Added”, “Modified”, or “Deleted”) files. Based on this list, the system removes 16 “Deleted” and 77 “Modified” files from the postings list and the term vocabulary, and then indexes only the 14 “Added” and 77 “Modified” files, instead of re-indexing all 6,186 source files present in version “602d549”. Therefore, when we evaluate subsequent bug reports, we only need to index the changed files. Furthermore, indexing is performed only once in case where multiple bugs are found in the same version of the project. This results in an average feature computation time that is much lower than the maximum time. Because the system updates the posting lists and the term vocabularies only for the changed files in the new commit, the average feature creation time, ranging from 18.11s to 279.28s, is representative for most bug reports.

The training time in Table 9 refers to the time needed to train the weight parameters of the SVM ranking function. The ranking time denotes the time needed to calculate the scoring function and then using it to rank all the source files for a bug report. As opposed to [70], the ranking time reported here does not include the time that it takes to read the feature vectors from disk. Overall, the runtime complexity at training and test time is dominated by the feature computation time.

9.2 Feature Complexity

Table 10 presents a breakdown of the time complexity and space complexity for every feature, as computed in the Eclipse Platform UI project. When the first bug report is processed, calculation of feature ϕ_1 requires that the system read all the source code files on disk, parse every source file into methods, perform tokenization and stemming, index all source files in method-level, and then calculate the lexical similarity between the bug report and every source file. This process takes 722.94 seconds, and needs 539.81 MB memory. For subsequent bug reports, the system needs to parse and index only the changed files. This leads to an average time of 8.51 seconds, which is much smaller than the maximum time needed for the first bug report. However, since the size of the posting list and the term vocabulary do not change much across bug reports, the average memory consumption stays close to the maximum. Therefore, we show only the maximum memory usage in the table.

Because the size of the API bag of words $m.api$ in Equation 5 is usually larger than the size of a method m in Equation 4, the calculation of ϕ_2 causes the largest time and memory usage. Features ϕ_3 to ϕ_{14} take less than 1 second to compute, on average, for a memory complexity less than 100 MB. Features ϕ_{15} to ϕ_{19} need between 1.57 to 7.06 seconds to compute, on average, and require about 300 MB of memory.

The system ranking accuracy and its runtime complexity can be traded-off against each other by combining the

TABLE 10
Time and memory usage for calculating every feature for all files on Eclipse Platform UI

Feature	Section	Short Description	Time (s)		Memory (MB)
			max	avg.	max
ϕ_1	3.1.1	Surface lexical similarity	722.94	8.51	539.81
ϕ_2	3.1.2	API-enriched lexical similarity	4046.62	44.07	683.34
ϕ_3	3.2	Collaborative filtering score	2.74	0.13	36.31
ϕ_4	3.3	Class name similarity	0.02	0.02	0.79
ϕ_5	3.4.1	Bug-fixing recency	0.02	0.02	1.76
ϕ_6	3.4.2	Bug-fixing frequency	0.02	0.02	1.81
ϕ_7	3.5	Summary-class names similarity	4.16	0.04	20.67
ϕ_8	3.5	Summary-method names similarity	11.85	0.21	65.79
ϕ_9	3.5	Summary-variable names similarity	34.58	0.48	55.07
ϕ_{10}	3.5	Summary-comments similarity	28.59	0.38	58.41
ϕ_{11}	3.5	Description-class names similarity	4.16	0.04	20.67
ϕ_{12}	3.5	Description-method names similarity	11.85	0.21	65.79
ϕ_{13}	3.5	Description-variable names similarity	34.58	0.48	55.07
ϕ_{14}	3.5	Description-comments similarity	28.59	0.38	58.41
ϕ_{15}	3.6.1	In-links = # of file dependencies of s	1.68	1.57	29,83
ϕ_{16}	3.6.1	Out-links = # of files that depend on s	1.68	1.57	29,83
ϕ_{17}	3.6.2	PageRank score	5.13	5.08	47.22
ϕ_{18}	3.6.3	Authority score	7.17	7.06	289.45
ϕ_{19}	3.6.3	Hub score	7.17	7.06	289.45

feature selection analysis from the previous section (e.g., Figure 19) with the runtime performance analysis done in this section (e.g., Table 10). For example, by using only features ϕ_1, ϕ_3, ϕ_4, and ϕ_7, the system achieves an MAP that is within 3% of the overall best MAP, while requiring significantly less time and space complexity. On the other hand, if a developer is interested in maximizing accuracy, using all the features will not require significant more time than using just the original 6 features. Because the file dependency features (ϕ_{15} to ϕ_{19}) are computed offline, the rest of the newly added features (ϕ_7 to ϕ_{14}) have an average runtime complexity that is negligible in comparison with the original 6 features.

10 Related Work

This paper builds on work that was previously reported in [70]. Novel contributions with respect to [70] include: the file dependency graph features (Section 3.6), incorporation of structural information retrieval features (Section 3.5), a detailed justification of the utility of fine-grained benchmark datasets for evaluation (Section 4), a description of the learning-to-rank optimization problem and the procedure used for tuning its hyperparameters (Section 6), updated experimental results that compare the improved LR approach against two additional state-of-the-art systems (Section 7), a new section on feature analysis that estimates the impact of each feature on the ranking accuracy (Section 8), and an expanded evaluation of runtime performance that breaks down the time and space complexity for every feature (Section 9).

10.1 Bug Localization

Recently, researchers have developed methods that concentrate on ranking source files for given bug reports automatically [30], [39], [52], [61], [62], [69], [71]. Saha et al. [62] syntactically parse the source code into four document fields: class, method, variable, and comment. The summary and the description of a bug report are considered as two query fields. Textual similarities are computed for each of the eight document field - query field pairs and then summed up into an overall ranking measure. Compared to our method, the approach from [62] assumes all features are equally important and ignores the lexical gap between bug reports and source code files. Furthermore, the approach is evaluated on a fixed version of the source code package of every project, which is problematic due to potential contamination with future bug-fixing information.

Kim et al. [30] propose both a one-phase and a two-phase prediction model to recommend files to fix. In the one-phase model, they create features from textual information and metadata (e.g., version, platform, priority, etc.) of bug reports, apply Naïve Bayes to train the model using previously fixed files as classification labels, and then use the trained model to assign multiple source files to a bug report. In the two-phase model, they first apply their one-phase model to classify a new bug report as either “predictable” or “deficient”, and then make predictions only for “predictable” reports. However, their one-phase model uses only previously fixed files as labels in the training process, and therefore cannot be used to recommend files that have not been fixed before when being presented with a new bug report. Furthermore, while their two-phase model aims at improving prediction accuracy by ignoring “deficient” reports, our approach can be used on all bug reports.

Zhou et al. [71] not only measure the lexical similarity between a new bug report and every source file but also give more weight to larger size files and files that have been fixed before for similar bug reports. Their model, namely BugLocator, depends only on one parameter α, even though it is based on three different features. The parameter is tuned on the same data that is used for evaluation, which means that the results reported in their paper correspond to training performance. It is therefore unclear how well their model generalizes to unseen bug reports. Wong et al. [69] show that source file segmentation and stack-trace analysis lead to improvements in BugLocator's performance. Our approach introduces more project-oriented features and learns the feature weights by training on previously submitted bug reports. The generalization performance is computed by running the trained model on a separate test dataset.

Nguyen et al. [52] apply LDA to predict buggy files for given bug reports. In their extended LDA model, the topic distribution of a bug report is influenced by the topic distributions of its corresponding buggy files. For ranking, they use the trained LDA model to estimate the topic distribution of a new bug report and compare it with the topic distributions of all the source files. They also introduce a defect-proneness factor that gives more weight to frequently fixed files and files with a large size. In evaluations conducted by other researchers [30], their approach performs comparably with the *Usual Suspects* method. While they model the training task as a classification problem in which bug reports and files are assigned to multiple topics, we directly train our model for ranking, which we believe is a better match for the way the model is used in testing.

Rao et al. [61] apply various IR models to measure the textual similarity between the bug report and a fragment of a source file. Through evaluations, they reported that more sophisticated models such as LDA and LSA did not outperform a Unigram model or VSM. Lukins et al. [39] combine LDA and VSM for ranking. They index source files with topics estimated by the LDA model, and use VSM to measure the similarity between the description of a bug report and the topics of a source file. Our approach builds relationships between bug reports and source files by extracting information not only from bug reports and code, but also from API documents and file dependencies in software repositories.

To support fault localization, a number of approaches [14], [16], [27], [37] use runtime information that was generated for debugging. Other approaches [13], [65] analyze dynamic properties of programs such as code changes in order to infer causes of failures. Jin and Orso [27], Burger and Zeller [14], and Cleve and Zeller [16] use passing and failing execution information to locate buggy program entities. Liu et al. [37] locate faults by performing statistical analysis on program runtime behavior. Unlike methods that require runtime executions, our approach responds to bug reports without the need to run the program.

10.2 Feature Location

The research on bug localization is within a broader area called concept or feature location, which associates human oriented concepts expressed in natural language, such as functionality requirements, with their implementation counterparts in code [8], [68].

Various IR approaches have been utilized in the area of feature location [21]. Marcus et al. [42] introduce a LSI-based model to locate code for feature descriptions provided by developers. Antoniol et al. [1], [2] collect method-level execution traces and rank methods based on the frequencies of method calls. Poshyvanyk et al. [56], [58] introduce PROMESIR, an approach that combines LSI and a scenario-based probabilistic ranking of method calls to locate bugs for Mozilla and Eclipse systems. The LSI-based ranking results are further clustered by incorporating Formal Concept Analysis (FCA) into the ranking model [55], [57]. Similar with PROMESIR, Liu et al. [38] combines LSI and execution traces, to rank only methods in the execution trace. Shepherd et al. [63] introduce an interactive approach to feature location that enables users to refine their queries. Gay et al. [24] combine an IR-based concept location method with explicit relevance feedback mechanisms to recommend artifacts for bug reports. Ashok et al. [3] introduce DebugAdvisor, a tool that allows search with "fat queries" that contain both structured and unstructured data describing a bug, whose output is a relationship graph that relates elements in the repository with bug descriptions and people.

API descriptions have been used before to bridge the lexical gap between user queries and code, in applications such as code search and traceability recovery. McMillan et al. [43] introduce a code search engine Exemplar that uses VSM to measure the lexical similarity between user queries and descriptions of API calls, such that code that contains API calls with higher similarity scores is ranked higher. Dasgupta et al. [18] propose a software traceability tool TraceLab, which expands code vocabularies by adding API descriptions for method calls. Bajracharya et al. [5] present a feature location technique called Structural Semantic Indexing (SSI), which expands the code with words from other code segments that contain the same API calls. The feature ϕ_2 in our system uses API descriptions in a different way: for each method, we concatenate the API descriptions of class and interface names associated with the explicit type declarations of all local variables. The similarities between method-level API descriptions and the bug report are then aggregated into the file-level feature ϕ_2.

Link analysis algorithms such as PageRank and HITS have also been used in the area of feature location. Dit et al. [22] apply the PageRank and HITS algorithms on a method-call graph extracted from runtime execution traces. The methods in the execution trace are first ranked using their LSI-based textual similarity with the query. The PageRank, Authority, and Hub scores are then used to prune the ranking results. McMillan et al. [44], [45] apply PageRank on a function-call graph extracted statically from the source code package and rank based on a combination of PageRank scores and the textual similarity with the query. Since our approach is targeted at ranking source code files, we apply link analysis algorithms on a different graph, the file dependency graph, and justify the PageRank score as a proxy for code complexity. In our fine-grained evaluation, we build a file-dependency graph for every before-fix commit, for a total of over 22,000 graphs.

Gethers et al. [25] use data mining of code changes to refine the ranking of methods for change requests (e.g., bug reports). Given a seed method related to the change request, they mine the project repository for methods that are co-changed with the seed method, in order to refine the IR-based ranking. The collaborative filtering feature ϕ_3 in our system determines whether a source file has been fixed for similar bug reports and thus exploits code changes in conjunction with the issue tracking history, without the need for seed data.

While Exemplar [43] uses a linear combination of features, it does not learn the weights for these features: all the weights are set manually to pre-defined values. Learning the weights of a large set of features (by exploiting previously solved bug reports) is a major factor in the competitive performance of our system, which follows the learning-to-rank approach to bug localization that we introduced earlier in [70]. A recent demo by Binkley and Lawrie [9] further illustrates the benefits of using the learning-to-rank technique in the context of feature location and traceability.

10.3 Defect Prediction

In contrast with the task of bug localization that requires bug reports as input, the task of defect prediction refers to predicting software defects before an abnormal behavior is found by users or developers. Notwithstanding this difference, defect prediction techniques can complement our approach by providing additional features quantifying the error-proneness of source code files. In order to support defect prediction, Lee et al. [36] analyze developer behaviors and build interaction patterns; Nagappan et al. [51] utilize the frequency of similar changes described as *change bursts*; Hassan et al. [26] use code change complexity; Zimmermann et al. [72] build a dependency graph that enables the computation of an error-proneness for files linked to buggy models. Moser et al. [48] and Kim et al. [31] use machine learning techniques to train a prediction model based on code changes. Kim et al. [32] cache fault-related code changes, and predict fault-prone entities based on the cached history. Menzies et al. [47] build a prediction model based on static code attributes. Nagappan et al. [50] apply Principal Component Analysis (PCA), while Bell et al. [6] and Ostrand et al. [53] use negative binomial regression to build models that predict fault-prone files.

11 Conclusion and Future Work

To locate a bug, developers use not only the content of the bug report but also domain knowledge relevant to the software project. We introduced a learning-to-rank approach that emulates the bug finding process employed by developers. The ranking model characterizes useful relationships between a bug report and source code files by leveraging domain knowledge, such as API specifications, the syntactic structure of code, or issue tracking data. Experimental evaluations on six Java projects show that our approach can locate the relevant files within the top 10 recommendations for over 70% of the bug reports in Eclipse Platform and Tomcat. Furthermore, the proposed ranking model outperforms three recent state-of-the-art approaches. Feature evaluation experiments employing greedy backward feature elimination demonstrate that all features are useful. When coupled with runtime analysis, the feature evaluation results can be utilized to select a subset of features in order to achieve a target trade-off between system accuracy and runtime complexity.

The proposed adaptive ranking approach is generally applicable to software projects for which there exists a sufficient amount of project specific knowledge, such as a comprehensive API documentation (Section 3.1.2) and an initial number of previously fixed bug reports (Section 6.1). Furthermore, the ranking performance can benefit from informative bug reports and well documented code leading to a better lexical similarity (Section 3.1.1), and from source code files that already have a bug-fixing history (Section 3.2).

In future work, we will leverage additional types of domain knowledge, such as the stack traces submitted with bug reports and the file change history, as well as features previously used in defect prediction systems. We also plan to use the ranking SVM with nonlinear kernels and further evaluate the approach on projects in other programming languages.

Acknowledgments

We would like to thank Hui Shen, who assisted us in setting up the experimental environment.

References

[1] G. Antoniol and Y.-G. Gueheneuc. Feature identification: A novel approach and a case study. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 357–366, Washington, DC, USA, 2005.

[2] G. Antoniol and Y.-G. Gueheneuc. Feature identification: An epidemiological metaphor. *IEEE Trans. Softw. Eng.*, 32(9):627–641, Sept. 2006.

[3] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala. Debugadvisor: A recommender system for debugging. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 373–382, New York, NY, USA, 2009.

[4] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 712–721, Piscataway, NJ, USA, 2013.

[5] S. K. Bajracharya, J. Ossher, and C. V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 157–166, New York, NY, USA, 2010.

[6] R. M. Bell, T. J. Ostrand, and E. J. Weyuker. Looking for bugs in all the right places. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 61–72, New York, NY, USA, 2006.

[7] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 308–318, New York, NY, USA, 2008.

[8] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, pages 482–498, Los Alamitos, CA, USA, 1993.

[9] D. Binkley and D. Lawrie. Learning to rank improves IR in SE. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 441–445, Washington, DC, USA, 2014.

[10] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet Allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.

[11] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann. Information needs in bug reports: Improving cooperation between developers and users. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, CSCW '10, pages 301–310, New York, NY, USA, 2010.

[12] B. Bruegge and A. H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[13] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 480–490, Washington, DC, USA, 2004.

[14] M. Burger and A. Zeller. Minimizing reproduction of software failures. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 221–231, New York, NY, USA, 2011.

[15] R. P. L. Buse and T. Zimmermann. Information needs for software development analytics. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 987–996, Piscataway, NJ, USA, 2012.

[16] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351, New York, NY, USA, 2005.

[17] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 433–436, New York, NY, USA, 2007.

[18] T. Dasgupta, M. Grechanik, E. Moritz, B. Dit, and D. Poshyvanyk. Enhancing software traceability by automatically expanding corpora with relevant documentation. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 320–329, Washington, DC, USA, 2013.

[19] H. Daumé, III and D. Marcu. A large-scale exploration of effective global features for a joint entity detection and tracking model. In *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*, HLT '05, pages 97–104, Stroudsburg, PA, USA, 2005.

[20] B. Dit, A. Holtzhauer, D. Poshyvanyk, and H. Kagdi. A dataset from change history to support evaluation of software maintenance tasks. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 131–134, Piscataway, NJ, USA, 2013.

[21] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[22] B. Dit, M. Revelle, and D. Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309, 2013.

[23] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 71–80, Washington, DC, USA, 2009.

[24] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the use of relevance feedback in IR-based concept location. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 351–360, 2009.

[25] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. Integrated impact analysis for managing software changes. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 430–440, Piscataway, NJ, USA, 2012.

[26] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, Washington, DC, USA, 2009.

[27] W. Jin and A. Orso. F3: Fault localization for field failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 213–223, New York, NY, USA, 2013.

[28] T. Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 133–142, New York, NY, USA, 2002.

[29] T. Joachims. Training linear SVMs in linear time. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 217–226, New York, NY, USA, 2006.

[30] D. Kim, Y. Tao, S. Kim, and A. Zeller. Where should we fix this bug? A two-phase recommendation model. *IEEE Trans. Softw. Eng.*, 39(11):1597–1610, Nov. 2013.

[31] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or Buggy? *IEEE Trans. Softw. Eng.*, 34(2):181–196, Mar. 2008.

[32] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 489–498, Washington, DC, USA, 2007.

[33] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, Sept. 1999.

[34] R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artif. Intell.*, 97(1-2):273–324, Dec. 1997.

[35] T. D. LaToza and B. A. Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 8:1–8:6, New York, NY, USA, 2010.

[36] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro interaction metrics for defect prediction. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 311–321, New York, NY, USA, 2011.

[37] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: Statistical model-based bug localization. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 286–295, New York, NY, USA, 2005.

[38] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 234–243, New York, NY, USA, 2007.

[39] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using Latent Dirichlet Allocation. *Inf. Softw. Technol.*, 52(9):972–990, Sept. 2010.

[40] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 03 1947.

[41] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[42] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 214–223, Washington, DC, USA, 2004.

[43] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Trans. Softw. Eng.*, 38(5):1069–1087, Sept. 2012.

[44] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 111–120, New York, NY, USA, 2011.

[45] C. Mcmillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Trans. Softw. Eng. Methodol.*, 22(4):37:1–37:30, Oct. 2013.

[46] P. Melville and V. Sindhwani. Recommender systems. In C. Sammut and G. Webb, editors, *Encyclopedia of Machine Learning*, pages 829–838. Springer US, 2010.

[47] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.*, 33(1):2–13, Jan. 2007.

[48] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, New York, NY, USA, 2008.

[49] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan. The design of bug fixes. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 332–341, Piscataway, NJ, USA, 2013.

[50] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 452–461, New York, NY, USA, 2006.

[51] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, pages 309–318, Washington, DC, USA, 2010.

[52] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 263–272, Washington, DC, USA, 2011.

[53] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4):340–355, Apr. 2005.

[54] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, The Stanford University InfoLab, November 1999.

[55] D. Poshyvanyk, M. Gethers, and A. Marcus. Concept location using formal concept analysis and information retrieval. *ACM Trans. Softw. Eng. Methodol.*, 21(4):23:1–23:34, Feb. 2013.

[56] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Softw. Eng.*, 33(6):420–432, June 2007.

[57] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ICPC '07, pages 37–48, Washington, DC, USA, 2007.

[58] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Gueheneuc, and G. Antoniol. Combining probabilistic ranking and latent semantic indexing for feature identification. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC '06, pages 137–148, Washington, DC, USA, 2006.

[59] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 432–441, Piscataway, NJ, USA, 2013.

[60] S. Ramaswamy, P. Tamayo, R. Rifkin, S. Mukherjee, C.-H. Yeang, M. Angelo, C. Ladd, M. Reich, E. Latulippe, J. P. Mesirov, T. Poggio, W. Gerald, M. Loda, E. S. Lander, and T. R. Golub. Multiclass cancer diagnosis using tumor gene expression signatures. *Proceedings of the National Academy of Sciences*, 98(26):15149–15154, 2001.

[61] S. Rao and A. Kak. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 43–52, New York, NY, USA, 2011.

[62] R. Saha, M. Lease, S. Khurshid, and D. Perry. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 345–355, Nov 2013.

[63] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th International Conference on Aspect-oriented Software Development*, AOSD '07, pages 212–224, New York, NY, USA, 2007.

[64] S. Shivaji, E. J. Whitehead, Jr., R. Akella, and S. Kim. Reducing features to improve code change-based bug prediction. *IEEE Trans. Softw. Eng.*, 39(4):552–569, Apr. 2013.

[65] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding failure-inducing changes in java programs using change classification. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 57–68, New York, NY, USA, 2006.

[66] E. M. Voorhees. The TREC-8 question answering track report. In *In Proceedings of TREC-8*, pages 77–82, 1999.

[67] A. W. Whitney. A direct method of nonparametric measurement selection. *IEEE Trans. Comput.*, 20(9):1100–1103, Sept. 1971.

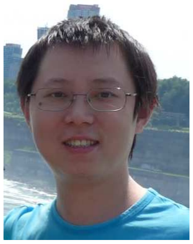
[68] N. Wilde, J. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 200–205, Nov 1992.

[69] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Proc. ICSME'14, To Appear*, 2014.

[70] X. Ye, R. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 689–699, New York, NY, USA, 2014.

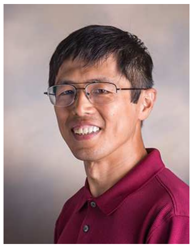
[71] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 14–24, Piscataway, NJ, USA, 2012.

[72] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 531–540, New York, NY, USA, 2008.



Xin Ye received the BSc and MS degree in electronic engineering from Beijing Institute of Technology, China. He is currently working toward the PhD degree in the School of Electrical Engineering and Computer Science at Ohio University, Athens, OH, USA. His research interests include search-based software engineering, mining software repositories, and program analysis. He is a student member of the IEEE.

Razvan Bunescu received a PhD degree in Computer Science from the University of Texas at Austin in 2007, with a thesis on machine learning methods for information extraction. He is currently an Associate Professor in Computer Science at Ohio University. His main research interests are in machine learning and computational linguistics, with a recent focus on applications in biomedical informatics, computer architecture, and software engineering.



Chang Liu Chang Liu received his PhD in Information & Computer Science from the University of California at Irvine in 2002. He is currently an Associate Professor in Computer Science at Ohio University. His current research focus is to turn Software Engineering Big Data into actionable knowledge for software developers and apply software engineering principles in learning and health applications.