Learning to Rank Relevant Files for Bug Reports using Domain Knowledge

Xin Ye, Razvan Bunescu, and Chang Liu
School of Electrical Engineering and Computer Science, Ohio University
Athens, Ohio 45701, USA
xy348709,bunescu,liuc@ohio.edu

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 of a project with respect to how likely they are to contain the cause of the bug would enable developers to narrow down their search and potentially could lead to a substantial increase in productivity. This paper introduces an adaptive ranking approach that leverages domain knowledge through functional decompositions of source code files into methods, API descriptions of library components used in the code, the bug-fixing history, and the code change history. Given a bug report, the ranking score of each source file is computed as a weighted combination of an array of features encoding domain knowledge, where the weights are trained automatically on previously solved bug reports using a learning-to-rank technique. We evaluated our 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 newly introduced learning-to-rank approach significantly outperforms two recent state-of-the-art methods in recommending relevant files for bug reports. 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.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging Aids*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Retrieval models*; I.2.6 [**Artificial Intelligence**]: Learning—*Parameter learning*; I.2.7 [**Artificial Intelligence**]: Natural Language Processing—*Text Analysis*

General Terms

Design, Experimentation, Languages

Permission to make digital or hard copies of all or part of this work for personal or

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00

Keywords

bug reports, software maintenance, learning to rank

1. INTRODUCTION AND MOTIVATION

A software *bug* or *defect* is a coding mistake that may cause unintended and unexpected behaviors of the software component [7]. 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 [10].

A developer who is assigned a bug report usually needs to reproduce the abnormal behavior [22] and perform code reviews [2] 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 [6]. Lexical mismatches between natural language statements in bug reports and technical terms in software systems [4] limit the accuracy of ranking methods that are based on simple lexical matching scores. To locate the bug, a developer needs to not only analyze the bug report using their domain knowledge, but also collect information from peer developers and users. Employing such a manual process in order to find and understand the cause of a bug can be tedious and time consuming [31]. Therefore, an automatic approach that ranked the source files with respect to their relevance for the bug report could speed up the bug finding process, which in turn will lead to an overall improvement in the software team productivity.

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) [27]. 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. Our system features 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. Furthermore, 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 [39]. 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 [28] 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.

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 strong 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 two 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 general enough to be applicable to a wide diversity of software projects for which domain knowledge, in the form of API documentation and syntactically parsed code, is readily available.

The main contributions of this paper include: using API descriptions to bridge the lexical gap between bug reports and source code; exploiting previously fixed bug reports as training examples for the proposed ranking model in conjunction with a learning-to-rank technique; and a strong benchmark dataset created by checking out a before-fix version of the source code package for each bug report.

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 strong benchmark datasets are introduced in Section 4, followed by a description of the experimental evaluation setting and results in Section 5. After a discussion of related work in Section 6, the paper ends with future work and concluding remarks.

2. RANKING MODEL

A ranking model is defined 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 ($k = 6$), where each feature $\phi_i(r, s)$ measures a specific relationship between the source file s and the received bug report r:

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

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 ENGINEERING

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:

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

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 [13, 41] 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, 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:

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

i.e., the maximum from all per-method similarities and the whole file similarity.

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 1 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 2 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 unuseful 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 ...

Figure 1: Eclipse bug report 339286.

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

Figure 2: Code from PartRenderingEngine.java

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 ...

Figure 3: API specification for `MUILabel` interface.

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 superinterface of `MTrimmedWindow`. As can be seen in Figure 3, the API documentation of the `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 extracts 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 superclasses or superinterfaces. For each method

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

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

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

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 ...

Figure 4: 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: ...

Figure 5: Bug reports that are similar with 378535.

then compute an API-enriched lexical similarity feature as follows:

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

i.e., the maximum from all per-method API similarities and the whole file API similarity.

3.2 Collaborative Filtering Score

It has been observed in [31] that a file that has been fixed before may be responsible for similar bugs. For example, Figure 4 displays an Eclipse bug report about incorrect menu options for parts that are not closeable. Figure 5 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, is is not surprising that source file *StackRenderer.java*, which had been previously found to be relevant for the three reports in Figure 4, was also found to be relevant for the textually similar bug report in Figure 5.

This *collaborative filtering* effect has been used before to improve the accuracy of recommender systems [28], 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:

$$\phi_3(r, s) = sim(r, br(r, s)) \quad (6)$$

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

3.3 Class Name Similarity

A bug report summary 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:

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

This feature will be automatically normalized during the feature scaling step described in Section 3.6.

3.4 Bug-Fixing Recency

The change history of source codes provides information that can help predict fault-prone files [39]. 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.

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 $last(r, s)$:

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

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.

3.5 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:

$$\phi_6(r, s) = |br(r, s)| \quad (9)$$

This feature will be automatically normalized during the feature scaling step described in Section 3.6 below.

3.6 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

Table 1: 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	
AspectJ	2002-03-13 – 2014-01-10	593	87	2	1	6,879	4,439	2,076	54
Birt	2005-06-14 – 2013-12-19	4,178	230	1	1	9,697	6,841	1,700	957
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
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

$\phi.min$ 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:

$$\phi' = \begin{cases} 0 & \text{if } \phi < \phi.min \\ \dfrac{\phi - \phi.min}{\phi.max - \phi.min} & \text{if } \phi.min \le \phi \le \phi.max \\ 1 & \text{if } \phi > \phi.max \end{cases} \tag{10}$$

4. BENCHMARK DATASETS

We created benchmark datasets for evaluation from six open-source projects:

1. AspectJ[4]: an aspect-oriented programming extension for Java.
2. Birt[5]: an Eclipse-based business intelligence and reporting tool.
3. Eclipse Platform UI[6]: the user interface of an integrated development platform.
4. JDT[7]: a suite of Java development tools for Eclipse.
5. SWT[8]: a widget toolkit for Java.
6. Tomcat[9]: a web application server and servlet container.

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 [12]. 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 [12]. Overall, we collected more than 22,000 bug reports from the six projects.

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. Consequently, using just one revision of the source code package for evaluation may lead to performance assessments that do not match the actual performance of the system when used in practice. For example, the fixed revision that is used for evaluation may contain future bug-fixing information for older bug reports. Furthermore, a buggy file might not even exist in the fixed revision, if it were deleted after the bug was reported. To avoid the problems associated with using a fixed code revision, we check out a before-fix version of the project for each bug report.

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 1 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 in our evaluation. Our dataset is publicly available[10].

5. EXPERIMENTAL EVALUATION

As described in Section 2, our 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. The model parameters w_i are trained using the learning-to-rank approach [17], as implemented in the SVM^{rank} package [18]. In this learning framework, the optimization procedure tries to find a set of parameters such that 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. Thus, if s_1 is known to be relevant for bug

[4]http://eclipse.org/aspectj/
[5]https://www.eclipse.org/birt/
[6]http://projects.eclipse.org/projects/eclipse.platform.ui
[7]http://www.eclipse.org/jdt/
[8]http://www.eclipse.org/swt/
[9]http://tomcat.apache.org

[10]http://dx.doi.org/10.6084/m9.figshare.951967

report r and s_2 is known to be irrelevant for the same bug report, then the objective of the optimization procedure is to find parameters w_i such that $f(r, s_1) > f(r, s_2)$. For any given bug report, the number of irrelevant source code files is very large, which would make the training time infeasible. Therefore, for each bug report r we first use the VSM cosine similarity feature $\phi_1(r, s)$ to rank all the files in the dataset and then select only the top 300 irrelevant files for training.

In order to create disjoint training and test data, 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 10 equally sized folds $fold_1$, $fold_2$, ..., $fold_{10}$, where $fold_1$ contains the most recent bug reports while $fold_{10}$ is the oldest. The reports from AspectJ are split only into 3 folds, due to the smaller size of the project. Furthermore, the oldest fold is split into 60% training and 40% validation, and a grid search is performed in order to tune the capacity parameter C of the ranking SVM. This is done by repeatedly training on the 60% and testing on the 40% for different values of C and selecting the one that maximizes mean average precision on the validation data. Since tuning on other folds results in similar values for C, we use the C value that was tuned on the oldest fold for all training folds.

The ranking model is trained on $fold_{k+1}$ and tested on $fold_k$, for all $k \leq 9$. Since the folds are arranged chronologically, this means that we always train on the most recent bug reports, which are supposed to better match the properties of the bugs in the current fold. For each bug report from a test fold, testing the model means computing the weighted scoring function $f(r, s)$ for each source code file using the learned weights, and 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 should be listed at the top. At the end, we pool the bug reports from all 9 test folds and compute the overall system performance using the following evaluation metrics:

- *Accuracy@k* measures the percentage of bug reports for which we make at least one correct recommendation in the top k ranked files.

- *Mean Average Precision (MAP)* is a standard metric widely used in information retrieval [27]. It is defined as the mean of the Average Precision (AvgP) values obtained for all the evaluation queries:

$$MAP = \sum_{q=1}^{|Q|} \frac{AvgP(q)}{|Q|}, \; AvgP = \sum_{k \in K} \frac{Prec@k}{|K|} \quad (11)$$

  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:

$$Prec@k = \frac{\text{\# of relevant docs in top k}}{k} \quad (12)$$

- *Mean Reciprocal Rank (MRR)* [43] is based on the position $first_q$ of the first relevant document in the ranked list, for each query q:

$$MRR = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \frac{1}{first_q} \quad (13)$$

5.1 Results and Comparisons

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 [19]

We also compared against 2 recent state-of-the-art systems:

1. BugLocator [45] ranks source files based on textual similarity, the size of source files, and information about previous bug fixes.

2. BugScout [34] classifies source files as relevant or not based on an extension to Latent Dirichlet Allocation (LDA) [5].

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 optimal value published in [45].

Figures 6 to 11 present the Accuracy@k results for the 4 implemented methods, with k ranging from 1 to 20. The LR approach achieves better results than the other three methods on all six projects. For example, on the Eclipse Platform UI our approach achieved Accuracy@k of 35.7%, 52.7%, 60.5%, and 70.4% for k = 1, 3, 5, and 10 respectively. That is to say if we recommend only one source file to users, we can make correct recommendations for 35.7% of 6,495 collected bug reports. If we recommend ten source files, we can make correct recommendations for 70.4% bug reports. In comparison, BugLocator achieved Accuracy@1 of 25.9% and Accuracy@10 of 59.7%. VSM and Usual Suspect achieved Accuracy@10 of 42.2% and 18.3%, respectively. An application of the Mann-Whitney U Test [26] shows that the LR approach significantly ($p < 0.05$) outperformed BugLocator in terms of Accuracy@k for Eclipse Platform UI, JDT, and SWT. It also significantly outperformed VSM and Usual Suspects in terms of Accuracy@k for all six projects.

AspectJ and Tomcat are two projects where BugLocator performs close to the LR approach. For many bug reports in AspectJ, it is often the case that relevant files are files that have been frequently fixed, which also explains the relatively high performance obtained by Usual Suspects. Since the bug-fixing information is exploited by both the LR approach and BugLocator, it is expected that they obtain comparable performance on this dataset. With respect to Tomcat, numerous bug reports contain rich descriptions that share many terms with the relevant files, which explains the relatively high performance obtained by VSM. Consequently, since both BugLocator and our LR approach exploit textual similarity between bug reports and source files, it is expected that they perform comparably on this dataset, too.

Figure 12 and Figure 13 compare the same 4 methods in terms of MAP and MRR. Here too, the LR approach outperforms the two baselines and BugScout on all six projects. For example, on the Eclipse Platform UI project, the MAP and MRR results for our LR approach are 0.40 and 0.47, which compare favorably with Bug Locator (0.31 and 0.37), VSM (0.20 and 0.25), and Usual Suspects (0.07 and 0.10).

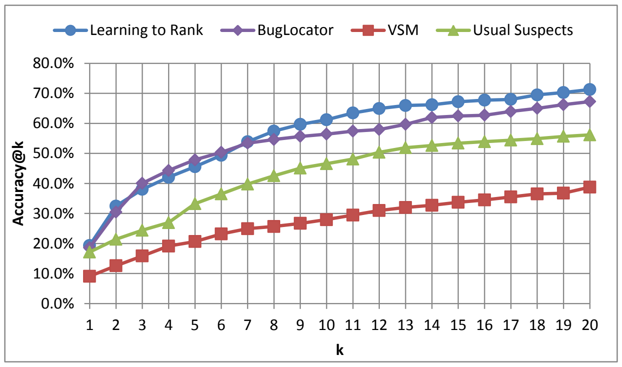


Figure 6: Accuracy graphs on AspectJ.

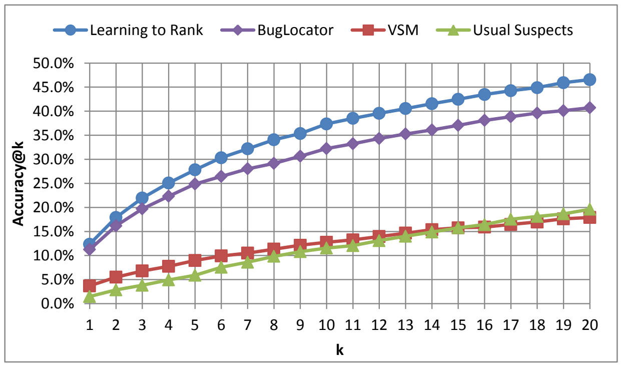


Figure 7: Accuracy graphs on Birt.

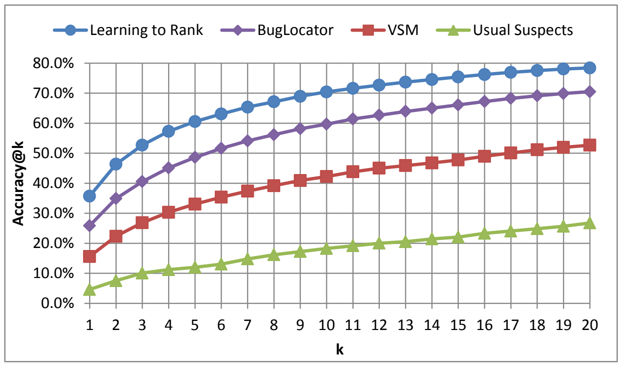


Figure 8: Accuracy graphs on Eclipse Platform UI.

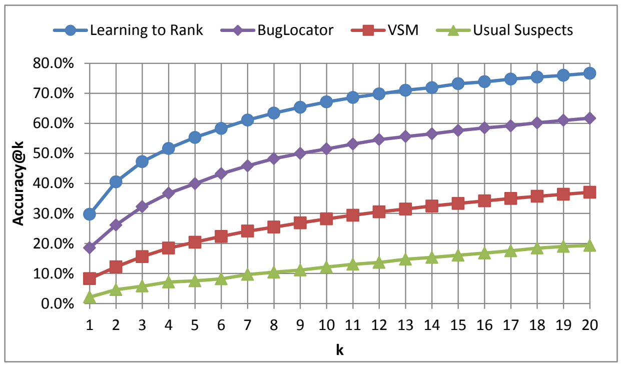


Figure 9: Accuracy graphs on JDT.

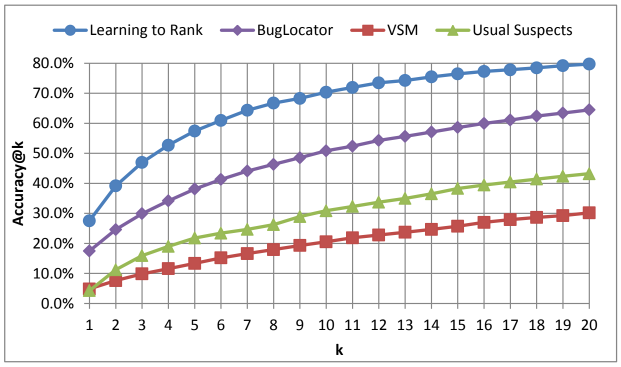


Figure 10: Accuracy graphs on SWT.

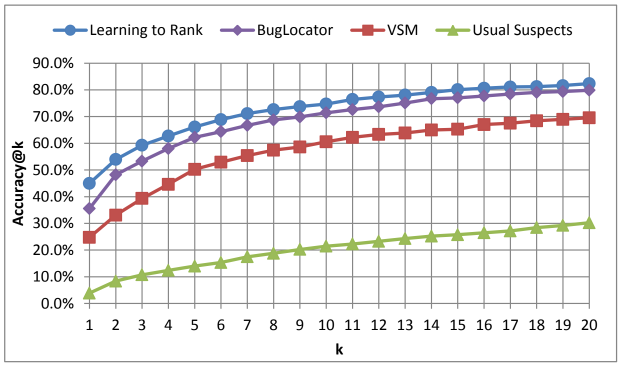
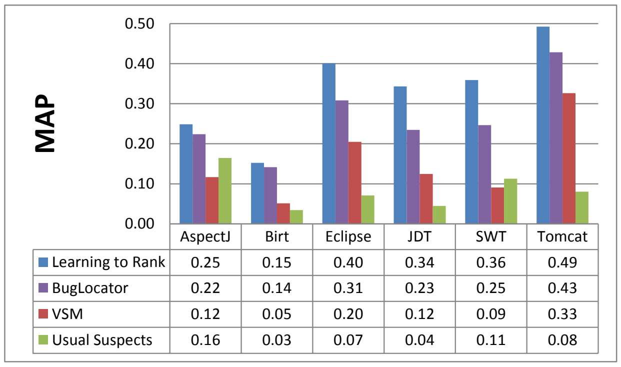
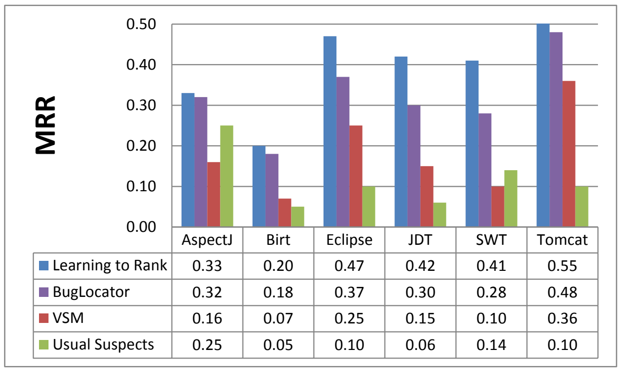


Figure 11: Accuracy graphs on Tomcat.



	AspectJ	Birt	Eclipse	JDT	SWT	Tomcat
Learning to Rank	0.25	0.15	0.40	0.34	0.36	0.49
BugLocator	0.22	0.14	0.31	0.23	0.25	0.43
VSM	0.12	0.05	0.20	0.12	0.09	0.33
Usual Suspects	0.16	0.03	0.07	0.04	0.11	0.08

Figure 12: MAP comparison.



	AspectJ	Birt	Eclipse	JDT	SWT	Tomcat
Learning to Rank	0.33	0.20	0.47	0.42	0.41	0.55
BugLocator	0.32	0.18	0.37	0.30	0.28	0.48
VSM	0.16	0.07	0.25	0.15	0.10	0.36
Usual Suspects	0.25	0.05	0.10	0.06	0.14	0.10

Figure 13: MRR comparison.

Table 2: Comparison between BugScout (BS) and Learning-to-Rank (LR) on data replicated from [34].

Project	Accuracy@1		Accuracy@10		Accuracy@20	
	BS	LR	BS	LR	BS	LR
AspectJ	<15%	23%	<40%	60%	<60%	73%
Eclipse	<15%	38%	<35%	72%	<40%	79%

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 [34] , we tried to recreate their dataset by following the description from [12, 34]. 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 BogScout was published). To train our LR method, we used bug reports that were solved before the bug reports in the testing dataset.

Table 2 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 [34] and are substantially lower than the LR results. While it is possible that our replicated dataset is different from their dataset, we believe there are two main reasons why the LR method performs better than BugScout:

1. LR uses features that capture domain knowledge relationships between bug reports and source files.
2. LR is trained directly to optimize ranking results. But BugScout is trained for classification into multiple topics, which may represent a mismatch during testing, when the system is evaluated for ranking.

Various IR approaches have been applied before on the task of identifying source files that are relevant for a bug report. These include approaches based on LDA [25, 34, 40], Latent Semantic Indexing (LSI) [40, 45], Smoothed Unigram Model (SUM) [40, 45], and SVMs [19, 34]. Since BugLocator was reported to outperform the existing approaches using LDA, LSI, and SUM [45], and since BugScout was reported to outperform the SVM model proposed in [34], we expect our LR system to compare favorably with all these previous approaches.

5.2 Evaluation of Feature Utility

In order to estimate the utility of the features used in our system, for each feature ϕ_i we report in Table 3 the corresponding weight w_i, averaged over all training folds $fold_k$, where $2 \leq k \leq 10$. Based on the magnitude of the weights, we can say for example that feature ϕ_1 is the most important feature in all projects but Birt, followed by feature ϕ_3. Looking in more detail at the Eclipse Platform UI project, we performed a set of evaluations in which we used each feature separately for ranking. The Accuracy@k results are shown in Figure 14 for each feature. The accuracy-based ranking of features shown in this figure is identical with the weight-based ranking from Table 3. The best results are achieved when the system uses all the features, a behavior that is consistent over all values of k from 1 to 20. The next best results are obtained when using only feature ϕ_1, followed by feature ϕ_3. That is to say, on this project, the most helpful information for ranking is provided by the textual similarity between the bug report and the source file and its methods, and by the textual similarity with the summaries of previously fixed bug reports.

Table 3: The average model parameters.

Project	w_1	w_2	w_3	w_4	w_5	w_6
AspectJ	10.63	3.92	5.79	2.29	4.04	4.26
Birt	2.28	3.84	8.20	1.27	1.61	3.57
Eclipse	7.77	3.47	6.23	0.82	0.86	4.39
JDT	15.64	5.07	4.11	0.24	0.04	0.52
SWT	16.93	1.23	4.99	1.32	1.16	10.48
Tomcat	14.00	3.69	6.38	1.45	0.64	1.67

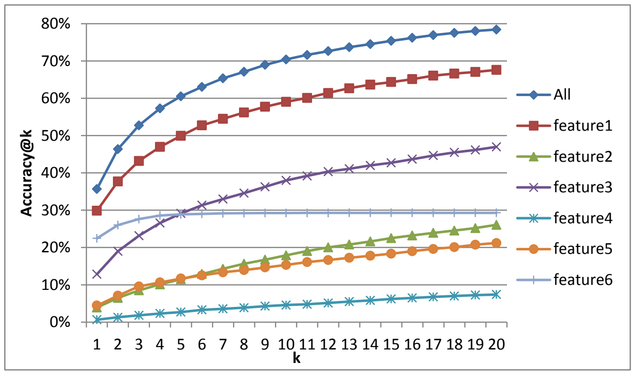


Figure 14: Single feature performance on Eclipse.

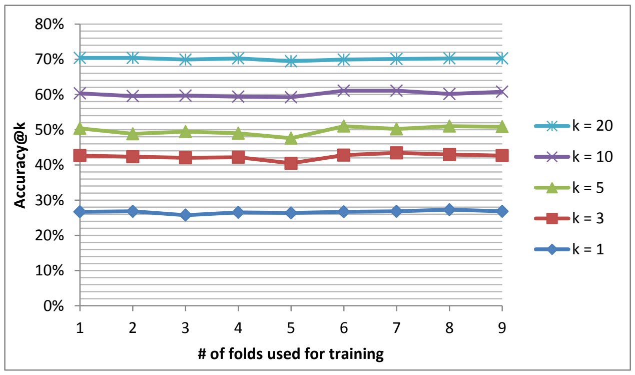


Figure 15: Learning Curves for Eclipse Platform UI.

5.3 Impact of Training Data Size

In the evaluation of the LR method, we always trained on $fold_{k+1}$ and tested on $fold_k$, for $k \leq 9$. To evaluate the impact of increasing the size of the training data, we ran an experiment on the Eclipse Platform UI project in which we kept the test dataset unchanged as $fold_1$, and trained on increasingly larger datsets. Thus, we first train on $fold_2$, then we trained on $fold_2 \cup fold_3$, until, in the final evaluation, we train on all 9 folds $fold_2 \cup fold_3 \cup ... fold_{10}$. The learning curves in Figure 15 show the behavior of Accuracy@k as a function of the number of folds used during training. The figure shows that increasing the size of the training data not improve the accuracy. In terms of MAP, training on $fold_2$

Table 4: Runtime Performance Results.

Project	index (s)		training (s)		ranking (s)	
	max	avg.	max	avg.	max	avg.
AspectJ	1300.78	17.59	1.51	1.51	0.28	0.14
Birt	2229.40	32.52	2.85	2.85	1.59	0.69
Eclipse	2577.48	44.26	3.32	3.32	1.19	0.63
JDT	3643.56	72.71	3.56	3.56	1.27	0.63
SWT	929.97	39.63	2.17	2.17	0.39	0.12
Tomcat	435.63	13.99	0.63	0.63	0.28	0.07

achieved a value of 0.3271, while the best result is 0.3299 when training on $fold_2$ to $fold_9$. The MRR when training on $fold_2$ is 0.3788, while the best MRR is 0.3834, obtained when training on $fold_2$ to $fold_9$. We applied the Mann-Whitney U Test and found no significant difference between the result of training on $fold_2$ and the result of training on a larger dataset.

Given the reduced number of parameters in our model, it is perhaps not surprising that we obtain flat learning curves. Furthermore, since $fold_2$ is chronologically the closest to $fold_1$, it is expected that bug reports from the other folds will not be as similar to bugs from $fold_1$, and thus not as useful for ranking source files with respect to these bugs. Furthermore, achieving optimal results using only a relatively small training dataset has a beneficial impact on the system's time and memory complexity.

5.4 Runtime Performance

We performed time complexity evaluations on a computer with CPU Intel(R) Core(TM) i7 920 2.67GHz (8 cores), 24G RAM, and Linux 3.2. Table 4 presents the runtime performance of our approach. The indexing time refers to the time used to create a posting list and a term vocabulary for source files, API descriptions, and previously fixed bug reports, respectively. The training time is the time needed to train the weight parameters of our ranking function. The ranking time is the time used to calculate the file scores and to rank all source files for a bug report. The average ranking time, ranging from 0.07s to 0.69s, makes the system suitable for practical use.

The maximum indexing time for every project is relatively high because we need to index all source files for the before-fix version of the first bug report. When using VSM, we need to index (calculate $tf_{t,d}$ and idf_t for) all source files and create a postings list and a term vocabulary [27]. To efficiently perform evaluation on over 22,000 before-fix project versions, we designed a method that indexes only the changed files. Taking the Eclipse bug 420972 as an example, we check out its before-fix version "2143203", index the 6,188 source files and perform evaluation. When we perform evaluation for another bug 423588, we check out its before-fix version "602d549" and use the `git diff` command to obtain the list of changed ("Added", "Modified", and "Deleted") files. Based on this list, we remove 16 "Deleted" and 77 "Modified" files from the postings list and the term vocabulary, and index only 14 "Added" plus 77 "Modified" files, instead of re-indexing all 6,186 source files present in version "602d549".

Therefore, when we evaluate another bug report, we only need to index the changed files. This results in an average indexing time that is much lower than the maximum index time. In practice, because we just need to update the posting lists and the term vocabularies for only the changed files in the new commit, the average indexing time, ranging from 13.99s to 72.71s, is representative for most cases. Furthermore, it is not necessary to perform indexing and training for every bug report because there may be multiple bugs found in the same version of the project.

6. RELATED WORK

Recently, researchers [19, 25, 34, 40, 41, 44, 45] have developed methods that concentrate on ranking source files for given bug reports automatically. Saha et al. [41] 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 [41] 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. [19] 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" report. 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. [45] 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. [44] show that source file segmentation and stack-trace analysis lead to complementary improvements in BugLocator's performance. In comparison, our approach introduces more project-oriented features and applies an automatic learning-to-rank technique to learn the weight of every feature on a separated training dataset. The generalization performance is computed by running the trained model on a separate test dataset.

Nguyen et al. [34] 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 large size. In evaluations conducted by other researchers [19], 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. [40] 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. [25] 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 software repositories.

To support fault localization, a number of approaches [9, 11, 16, 24] use runtime information that was generated for debugging. Other approaches [8, 42] analyze dynamic properties of programs such as code changes in order to infer causes of failures. Jin and Orso [16], Burger and Zeller [9], and Cleve and Zeller [11] use passing and failing execution information to locate buggy program entities. Liu et al. [24] locate faults by performing statistical analysis on program runtime behavior. Unlike these methods that require runtime executions, our approach responds to bug reports without the need to run the program.

Other researchers build models that associate bug reports to individual developers and functions, in addition to source files. Ashok et al. [1] introduce DebugAdvisor, a tool that allows search with free-text queries that contain both structured and unstructured data describing a bug. A graph with linked elements in the repository is used to recommend files and functions. Gay et al. [14] combine an IR-based concept location method with explicit relevance feedback mechanisms to recommend artifacts for bug reports. Poshyvanyk et al. [37, 38] introduce PROMESIR, which combines LSI and execution scenario based probabilistic ranking method, to locate bugs for Mozilla and Eclipse systems.

Another related area focuses on predicting software defects. In order to support defect prediction, Lee et al. [23] analyze developer behaviors and build interaction patterns; Nagappan et al. [33] utilize the frequency of similar changes described as *change burts*; Hassan et al. [15] use code change complexity; Zimmermann et al. [46] build a dependency graph that implies an error-proneness for files linked to buggy models. Moser et al. [30] and Kim et al. [20] use machine learning techniques to train a prediction model based on code changes. Kim et al. [21] cache fault-related code changes, and predict fault-prone entities based on the cached history. Menzies et al. [29] build a prediction model based on static code attributes. Nagappan et al. [32] apply principle component analysis (PCA), while Bell et al. [3] and Ostrand et al. [35] use negative binomial regression to build models that predict fault-prone files.

7. CONCLUSION & 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, and 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. Our ranking model outperforms BugLocator [45] and BugScout [34], two recent state-of-the-art approaches.

In future work, we will leverage additional types of domain knowledge, such as the authorship of a source file or the PageRank [36] score associated with each file in the dependency graph of the project. We also plan to use the ranking SVM with nonlinear kernels. The model will be further evaluated on projects in other programming languages.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

[1] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala. DebugAdvisor: A recommender system for debugging. In *Proc. ESEC/FSE '09*, pages 373–382, 2009.

[2] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. ICSE '13*, pages 712–721, 2013.

[3] R. M. Bell, T. J. Ostrand, and E. J. Weyuker. Looking for bugs in all the right places. In *Proc. ISSTA '06*, pages 61–72, 2006.

[4] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proc. SIGSOFT '08/FSE-16*, pages 308–318, 2008.

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

[6] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann. Information needs in bug reports: Improving cooperation between developers and users. In *Proc. CSCW '10*, pages 301–310, 2010.

[7] 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.

[8] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *Proc. ICSE '04*, pages 480–490, 2004.

[9] M. Burger and A. Zeller. Minimizing reproduction of software failures. In *Proc. ISSTA '11*, pages 221–231, 2011.

[10] R. P. L. Buse and T. Zimmermann. Information needs for software development analytics. In *Proc. ICSE '12*, pages 987–996, 2012.

[11] H. Cleve and A. Zeller. Locating causes of program failures. In *Proc. ICSE '05*, pages 342–351, 2005.

[12] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proc. ASE '07*, pages 433–436, 2007.

[13] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Proc. MSR '09*, pages 71–80, 2009.

[14] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the use of relevance feedback in IR-based concept location. In *Proc. ICSM '09*, pages 351–360, 2009.

[15] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proc. ICSE '09*, pages 78–88, 2009.

[16] W. Jin and A. Orso. F3: Fault localization for field failures. In *Proc. ISSTA '13*, pages 213–223, 2013.

[17] T. Joachims. Optimizing search engines using clickthrough data. In *Proc. KDD '02*, pages 133–142, 2002.

[18] T. Joachims. Training linear SVMs in linear time. In *Proc. KDD '06*, pages 217–226, 2006.

[19] 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.

[20] 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.

[21] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proc. ICSE '07*, pages 489–498, 2007.

[22] 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, 2010.

[23] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro interaction metrics for defect prediction. In *Proc. ESEC/FSE '11*, pages 311–321, 2011.

[24] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: Statistical model-based bug localization. In *Proc. ESEC/FSE-13*, pages 286–295, 2005.

[25] 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.

[26] 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.

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

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

[29] 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.

[30] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proc. ICSE '08*, pages 181–190, 2008.

[31] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan. The design of bug fixes. In *Proc. ICSE '13*, pages 332–341, 2013.

[32] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. ICSE '06*, ICSE '06, pages 452–461, 2006.

[33] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Proc. ISSRE '10*, pages 309–318, 2010.

[34] 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 *Proc. ASE '11*, pages 263–272, 2011.

[35] 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.

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

[37] 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.

[38] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Gueheneuc, and G. Antoniol. Combining probabilistic ranking and Latent Semantic Indexing for feature identification. In *Proc. ICPC '06*, pages 137–148, 2006.

[39] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proc. ICSE '13*, pages 432–441, 2013.

[40] S. Rao and A. Kak. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *Proc. MSR '11*, pages 43–52, 2011.

[41] R. Saha, M. Lease, S. Khurshid, and D. Perry. Improving bug localization using structured information retrieval. In *Proc. ASE'13*, pages 345–355, 2013.

[42] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding failure-inducing changes in java programs using change classification. In *Proc. SIGSOFT '06/FSE-14*, pages 57–68, 2006.

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

[44] 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.

[45] 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 *Proc. ICSE '12*, pages 14–24, 2012.

[46] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proc. ICSE '08*, pages 531–540, 2008.