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

**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 percent of the bug reports in the Eclipse Platform and Tomcat projects.

**EXISTING SYSTEM:**

* Recently, researchers have developed methods that concentrate on ranking source files for given bug reports automatically.
* Saha et al. 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.
* Kim et al. 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 Na€ıve 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.
* Rao and Kak apply various IR models to measure the textual similarity between the bug report and a fragment of a source file.

**DISADVANTAGES OF EXISTING SYSTEM:**

* 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.
* Existing methods require runtime executions.

**PROPOSED SYSTEM:**

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.

**ADVANTAGES OF PROPOSED SYSTEM:**

* Our approach can locate the relevant files within the top 10 recommendations for over 70 percent 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.

**SYSTEM ARCHITECTURE:**



**MODULES:**

* System Construction Module
* Ranking Function
* Feature Representation
* Collaborative Filtering Score

**MODULES DESCRIPTION:**

**System Construction Module:**

* In the first module, we develop the system with the entities required to evaluate our proposed model. 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. So In 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 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 project we apply three entities namely User, Developer, Admin. If User has an error in a source code then user send the error message to the Admin. Then Admin analysis the errors and ranking the reports and send to the Developers. And Developers find the solutions of the errors.

**Ranking Function:**

* 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.
* 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.
* 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.

**Feature Representation:**

* The proposed ranking model requires that a bug report - source file pair (r,s) be represented as a vector of k features. We distinguish between two major categories of features.
* Query dependent: These are features 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.

**Collaborative Filtering Score:**

* It has been observed in that a file that has been fixed before may be responsible for similar bugs. This collaborative filtering effect has been used before in other domains to improve the accuracy of recommender systems, 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 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.
* In a 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.
* In a File Revision History the source code change history provides information that can help predict fault-prone files. 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.

**SYSTEM REQUIREMENTS:**

**HARDWARE REQUIREMENTS:**

* System : Pentium Dual Core.
* Hard Disk : 120 GB.
* Monitor : 15’’ LED
* Input Devices : Keyboard, Mouse
* Ram : 1GB.

**SOFTWARE REQUIREMENTS:**

* Operating system : Windows 7.
* Coding Language : JAVA/J2EE
* Tool : Netbeans 7.2.1
* Database : MYSQL

**REFERENCE:**

Xin Ye, Student Member, IEEE, Razvan Bunescu, and Chang Liu, Senior Member, IEEE, “Mapping Bug Reports to Relevant Files: A Ranking Model, a Fine-Grained Benchmark, and Feature Evaluation”, **IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 42, NO. 4, APRIL 2016.**