HAL
archives-ouvertes.fr

Bug Localization Using Revision Log Analysis and Open Bug Repository Text Categorization

Amir H. Moin, Mohammad Khansari

▶ To cite this version:

Amir H. Moin, Mohammad Khansari. Bug Localization Using Revision Log Analysis and Open Bug Repository Text Categorization. 6th International IFIP WG 2.13 Conference on Open Source Systems,(OSS), May 2010, Notre Dame, United States. pp.188-199, 10.1007/978-3-642-13244-5_15 . hal-01056055

HAL Id: hal-01056055
https://hal.inria.fr/hal-01056055

Submitted on 14 Aug 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution| 4.0 International License

Bug Localization Using Revision Log Analysis and Open Bug Repository Text Categorization

Amir H. Moin and Mohammad Khansari

Department of IT Engineering, School of Science & Engineering, Sharif University of Technology, International Campus, Kish Island, Iran
moin@kishlug.ir, khansari@sharif.edu

Abstract. In this paper, we present a new approach to localize a bug in the software source file hierarchy. The proposed approach uses log files of the revision control system and bug reports information in open bug repository of open source projects to train a Support Vector Machine (SVM) classifier. Our approach employs textual information in summary and description of bugs reported to the bug repository, in order to form machine learning features. The class labels are revision paths of fixed issues, as recorded in the log file of the revision control system. Given an unseen bug instance, the trained classifier can predict which part of the software source file hierarchy (revision path) is more likely to be related to this issue. Experimental results on more than 2000 bug reports of 'UI'component of the Eclipse JDT project from the initiation date of the project until November 24, 2009 (about 8 years) using this approach, show weighted precision and recall values of about 98% on average.

1 INTRODUCTION

Both the total number of open source software projects and the total amount of open source code in the world, are growing at an exponential rate[1]. In addition, the number of developers interested in working in this field, are increasing tremendously fast. For example, the number of developers involved in the Linux kernel development project has doubled over the past three years[2]. Hence, one should expect a very high rate of bug reporting to the issue tracking system of large open source projects. As an example, consider the case of the Eclipse open bug repository, with an average bug reporting rate of above 50 issues per day from January 1 until November 24, 2009[3]. Suppose that each issue takes an average of ten minutes from a developer in order to be localized in the software source file hierarchy. This simply means, at least 8 professional person-hours per day is required merely for searching where the buggy piece of the code is located, which is indeed an invaluable and rare resource for most open source projects.

In this paper, we present a new approach for automating bug localization, i.e. finding the most relevant part of the software source file hierarchy to a bug reported to an open bug repository. Firstly, we analyze the history of source revisions, available in the log file of the version control system, in order to find the bug IDs and their corresponding revision path (path of the revised file during a successful bug fix). Secondly, we send a query to the open bug repository of the project in order to obtain summary and description of the extracted bug IDs. Then, we prepare our dataset in the proper and acceptable format for training the classifier. Afterward, we perform the classification via training a Support Vector Machine (SVM) classifier. Finally, given a new bug, we can localize the bug in the software source file hierarchy using the trained classifier.

Our approach is novel in that we use the large amount of valuable information in the open bug repositories of open source projects rather than performing analysis on the software source repositories to find latent software defects[14][15][16][17][18][19][20][21]. Moreover, rather than looking for an exact bug-related source file and its line number, we localize the bug in one higher level of the software source file hierarchy (file path). One of the possibly useful applications of this approach could be in bug triage, i.e. deciding each reported issue should be assigned to which developer in order to be fixed[4] in a time and cost effective manner. In that problem, the triager could assign bugs with respect to the field of expertise[5] and level of interest[6] of developers in that particular part of the software source file hierarchy.

The paper is organized as follows. Section 2 provides some background about revision control systems, open bug repositories and machine learning. In section 3, we present the proposed approach for bug localization. Section 4 provides validation and experimental results, and Section 5 reviews related work. Finally, we draw conclusion and suggest future work in section 6.

2 BACKGROUND

To understand the proposed approach one should be familiar with various areas including version control systems, open bug repositories and machine learning. We review the related concepts of these topics by giving examples from Eclipse projects.

2.1 Version Control Systems

A Version Control System (or more accurately, revision control system) is a combination of technologies and practices for tracking and controlling changes to a project's files, in particular to source code, documentation, and web pages. The main role of such a system is change management via identifying each change to the project, annotating it with relevant metadata such as the date, author, and possibly the reason of that change, and finally replaying these facts

to whoever asks, in the desired format. In other words, it is an inter-developer communication mechanism where a change is the basic unit of information. The most widely used revision control system in the Free[1]/Open Source Software (FOSS) world is Concurrent Versions System (CVS). Although it has become the default choice along the time and most experienced developers are already familiar with CVS, it has few disadvantages which consequently has led to the emergence of a number of alternatives such as Subversion (SVN), Git, Bazaar, and Mecurial[7].

Fortunately, the log files of the version control system for different components of a software project could be queried and saved in separate files. Figure 1 shows a very small part of the CVS log file for 'Core'component of the Eclipse JDT project.

```
RCS file:
/cvsroot/eclipse/org.eclipse.jdt.core/compiler/org/eclipse/jdt/inte
rnal/compiler/parser/RecoveredField.java,v
...
revision 1.39.2.1
date: 2009-06-19 15:38:30 +0430;  author: daudel;  state: Exp;
lines: +20 -0;  commitid: fb0c4a3b71ac4567;
R3_5_maintenance - Bug 277204
```

Fig. 1. A small part of a sample CVS log, the Eclipse JDT Project. ('...' represents the omitted lines.)

2.2 Open Bug Repositories

Providing a bug tracking system (or more accurately, issue tracking system) is one of the necessary tools of open source software development[7]. A bug tracking system usually consists of a database known as bug repository which contains information about the bug reports. Almost any open source project is supported by an open bug repository in which anyone could have a username and password and either report an issue or put a comment on an existing report.

There are various bug tracking software such as Bugzilla and JIRA. Furthermore, some projects like Debian GNU/Linux have their own bug tracking system[8].

[1]Here, Free is a matter of liberty, not a matter of price. For more information please visit http://www.gnu.org/philosophy/free-sw.html

Structure of Bug Reports

One of the better known bug tracking systems is Bugzilla. A typical bug report in Bugzilla consists of various parts including the predefined fields, free-form text, attachments and dependencies[4].

Figure 2 depicts the predefined fields in a sample bug report of the Eclipse bug repository. This bug report corresponds to the CVS log shown in figure 1. Some predefined fields such as the bug ID or reporter are specified when the report is created and fixed over the life cycle of the bug report (this life-cycle is covered in the next subsection). Other fields, either change successively while the bug report is tossed among the developers, i.e. forwarded from the developer to whom it is initially assigned to another one[9], like the Assigned To field, or change occasionally such as the Importance or the CC list[2][10].

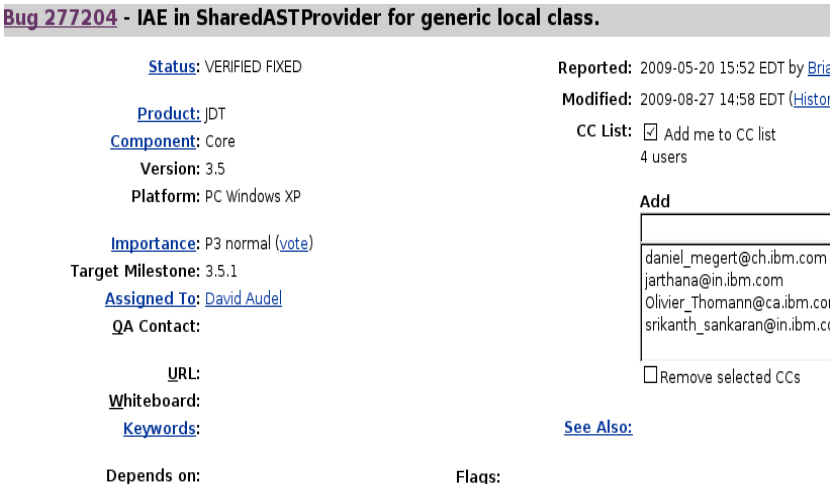

Fig. 2. Predefined fields in a sample Bugzilla bug report

The free-form text includes a one line summary of the issue, also known as its title, a detailed description of the report which should help a developer reproduce the bug and finally a number of comments on this issue which might refer to other similar bugs[4].

Other parts of bug reports include attachments and dependencies. Attachments are usually non-textual information such as screen-shots. Moreover, the bug tracking system tracks bugs which their resolution depend on fixing a specific bug report[10].

[2]The CC list is the list of the email addresses of people who are interested to be kept up-to-date about the status of the issue.

Life-cycle of Bug Reports

Initially, when a new issue comes to the open bug repository of the Eclipse projects, its status field is set to NEW. Then either it is assigned to a developer by the triager or a volunteer developer accepts its responsibility. Consequently, it is tagged with ASSIGNED.[3] At the end, when there is no remaining task due to the resolution of the bug report, it is marked as RESOLVED. If the triager finds that this issue is already reported, it is marked as RESOLVED DUPLICATE. If the report is not indeed a bug report, for example it states a natural feature of the software which is mistakenly thought to be a bug, the report is tagged with RESOLVED INVALID. When the erroneous behavior is not repeatable, perhaps because of poor description of the problem, the developer sets the status to RESOLVED WORKSFORME. Otherwise, the resolution might need applying changes in the source code which causes the issue to be marked as RESOLVED FIXED. If a bug is believed to be unsolvable for any reason, it will be tagged with RESOLVED WONTFIX[11].

The resolution status of the RESOLVED reports may later change to VERIFIED and then CLOSED. One is allowed to reopen a previously RESOLVED, VERIFIED or even CLOSED issue at any time. Figure 3 shows the typical life-cycle of the Eclipse bug reports[11].

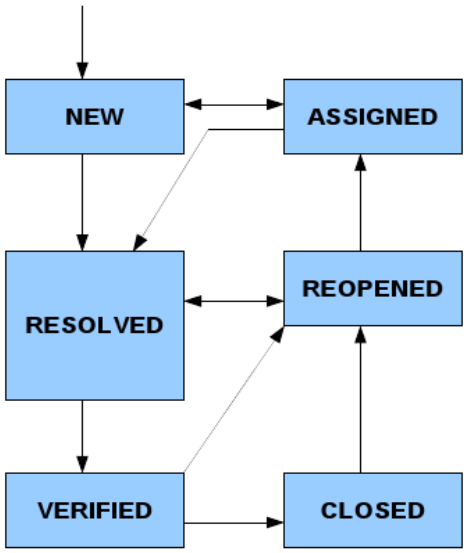

Fig. 3. Life-cycle of bug reports in Eclipse projects

[3]There exist few cases in which bug reports are not assigned to developers and resolved immediately by the triager.

In this paper, we only care about the bug reports which are either RESOLVED FIXED, VERIFIED FIXED or CLOSED FIXED.

2.3 Machine Learning

Machine learning is a discipline concerned with design and development of algorithms in order to allow computers to learn how to recognize complex patterns in data, to be able to make smart decisions. In the context of machine learning, the training data consist of a number of examples which are called instances. Each instance bears a number of input objects known as attributes or features which are usually encapsulated in a vector. In supervised machine learning an output value is assigned to each instance of the training data in advance and the problem is to deduce a function in order to predict the output value of any similar valid input vector. If the output value is a continuous value, the problem is called regression; otherwise, the output value is called the class label, the function is named as classifier and the problem is called classification. One of the many applications of this kind of classification is in text categorization, where the classifier is expected to assign a relevant category to an arbitrary text document based on a number of previously seen examples[12][13].

3 THE PROPOSED APPROACH

Given a new bug report from the open bug repository of an open source software project, our approach uses a Support Vector Machine (SVM) classifier to suggest the part of the source file hierarchy which is more likely to be related to this issue. The suggestion is made based on a number of previously seen examples, i.e. fixed bug reports in the past. Various components engaged in the proposed approach are presented in figure 4.

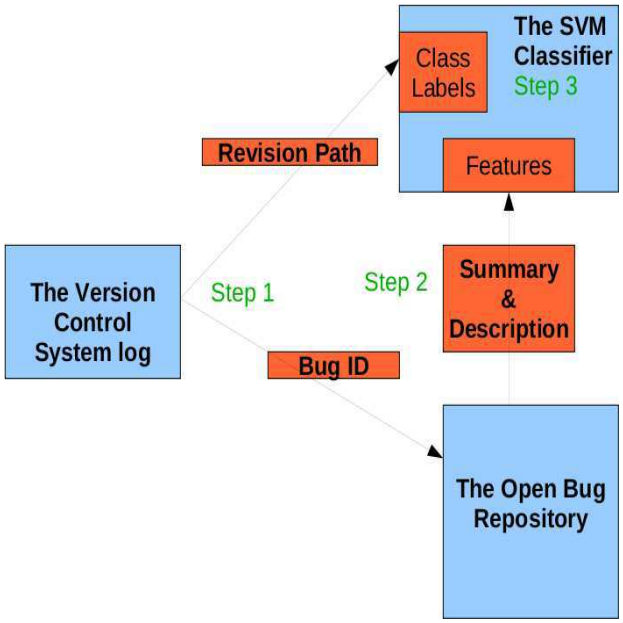

Fig. 4. Various components of the proposed approach

Our approach has three steps:

1. Analyzing the Revision Logs

   When a bug is fixed by a developer, the revision path, i.e. the path of the file which is revised due to this bug resolution, is not mentioned anywhere in the open bug repository. Consequently, one should analyze the entire change history of a specific software component in the log file of the version control system for that component, in order to find patterns such as 'fix for bug no ...'or similar among the comments of developers. The extracted bug IDs are used in step two, and the revision paths are used as class labels of the classifier in step three.

2. Querying the Bug Repository

   For each extracted bug ID in the previous step, we send a query to the bug tracking system and ask for the summary and description of that bug ID.

3. Training the Classifier

   After conducting previous steps, we have the revision path in software file hierarchy as well as the summary and description for each resolved bug ID. We use this information to train a Support Vector Machine (SVM) classifier.

4 VALIDATION & EXPERIMENTAL RESULTS

We have trained and tested our classifier with fixed[4] bug reports in the open bug repository of the Eclipse JDT Project ('UI'component), reported from the initiation date of the project until November 24, 2009. Thus, we have worked with more than 2000 bug reports.

We analyze the revision logs through the aid of a couple of useful GNU/Linux (and UNIX) commands, grep and awk. The result of this step is expected to be a number of textual files, each named with an existing path (directory level rather than file level) in the source file hierarchy and filled with all bug IDs related to that specific path.

After analyzing the CVS log file of the software component, 23 revision paths were found. A few number of these paths as well as the number of bug reports related to each, are shown in table 1.

Table 1. Several revision paths of 'UI'component of the Eclipse JDT project

Revision paths	No. of bugs
ui/org/eclipse/jdt/internal/ui/	1392
core extension/	133
core refactoring/	137
ui/org/eclipse/jdt/ui/	203

We have developed a Java application in order to connect to the Bugzilla open bug repository of the Eclipse JDT project using XML Remote Procedure Call (XML-RPC), a well known protocol for performing remote procedure calls over HTTP. For each of the bug IDs gathered in the previous step, we send a query to the bug tracking system and ask for the summary and description of that bug report. Eventually, we save the collected information about each bug in a separate textual file, named the same as the bug ID. One should keep every file related to a specific revision path in a distinct directory which is named after the revision path, in order to create a dataset to be used in the following steps.

In order to implement our approach, we use the Free/Open Source Software (FOSS) suite for machine learning written in Java, called WEKA. WEKA requires both the training and testing datasets to be in a standard format called ARFF. Fortunately, there is a converter, named TextDirectoryLoader in WEKA. This converter, receives a number of directories which contain a set of text files, and then treats the directory names as class labels, the text files as instances of each class and the information within each text file as features of that instance. The output of this converter is an ARFF file as desired.

[4]Trivially, the revision path for unfixed bugs is meaningless.

Since the classifier which we use in the next step cannot handle String attributes, we must apply an appropriate filter to the dataset, i.e. the ARFF file, in order to perform TF-IDF (Term Frequency-Inverse Document Frequency) transformation. This transformation is often used in information retrieval and text mining problems in order to give a weight to each term, based on the number of occurrence of the term. The basic assumption is that the more times a specific term appears in a text document, the more important it is to that document[25]. There is a filter in WEKA, called StringToWordVector which does the needed transformation easily. The output is still an ARFF file.

The classifier also cannot handle numeric attributes. However, our ARFF file contains a number of such attributes. The solution is applying another filter available in WEKA, named NumericToNominal. Now, the resulted ARFF file is ready to be used for training the SVM classifier.

After gathering and preparation of the dataset, the next step is to train the classifier and validate the learned model.

We use an improved Support Vector Machines (SVMs) algorithm, called Sequential Minimal Optimization (SMO) with linear kernel. SMO is much faster and more memory-efficient than the initial SVM algorithm[26][27]. We use binary SMO implementation with linear kernel which is available in WEKA as BinarySMO. This implementation replaces all missing values and transforms nominal attributes into binary ones. It also normalizes all attributes by default. The multi-class problem is solved by using pairwise classification[28]. Table 2 shows several normalized attribute weights of our dataset.

Table 2. Several normalized attribute weights

Attribute(term)	Weight
JavaCore	0.0847
Synchronizer	-0.0328
WM_CHAR	0.0173
container	0

As in any other machine learning problem, we should somehow evaluate the performance of our approach. We use ten fold cross validation for training and validation of the linear SVM classifier. The detailed evaluation results are provided in table 3.

The True Positive (TP) rate is equivalent to Recall. It measures how much part of the class is captured. In other words, the TP rate (Recall) is the proportion

Table 3. Detailed evaluation results of the binary SMO classifier

TP Rate	FP Rate	Precision	Recall	F-Measure	Class
0.992	0.062	0.99	0.992	0.991	0
0.938	0.008	0.951	0.938	0.944	1
0.985	0.055	0.985	0.985	0.965	Weighted Avg.

of the instances which are classified as class A, among all instances which indeed have class A.

The False Positive (FP) rate is the proportion of the instances which are classified as class A, but belong to a different class, among all instances which are not of class A.

The Precision is the proportion of the instances which indeed have class A, among all those instances which are classified as class A.

Since, often there is a trade-off between precision and recall, it is common to measure the classification performance via a mixture of both, called F-Measure[29].

$F - Measure = \frac{2*Precision*Recall}{Precision+Recall}$

Finally, Accuracy is the proportion of the total number of correctly classified instances among all instances. Our accuracy through the classification has been 98.5137%.

5 RELATED WORK

We are aware of a number of valuable efforts in the field of bug localization automation. One possible approach is trying to find bugs through checking either a well-specified program model[14] or real code directly[15][16] within the software source code. This approach is called static analysis[17].

Gyimothy et al.[18] use two groups of machine learning algorithms, decision trees and neural networks to predict buggy classes with a static code analysis approach.

The second approach is called dynamic analysis which is concerned with the comparison of the run-time behavior of correct and incorrect executions in order to localize suspicious segments of the source code[19][20]. This approach only labels program executions as correct or incorrect and needs no prior knowledge of the semantics of the software project[17].

Brun and Ernst[21] use Ernst's Daikon dynamic invariant extractor[22] to capture invariant features from the software source code with known errors and with errors removed. Then two groups of machine learning algorithms, Support Vector Machines (SVMs) and decision trees are employed to classify invariants as either fault-invariant or non-fault-invariant.

Most recently, Kim et al.[23][24] has proposed a new technique for predicting latent software bugs, called change classification. They use Support Vector Machines (SVMs) to predict whether a specific change to the software source is more likely to be buggy or clean, based on the previous change history.

Kim et al.'s approach is similar to ours in a couple of aspects. Firstly, they analyze log files of the version control system of software projects to find related bug fixes in order to label that change in the source code as buggy. Similarly, we analyze those files in order to find bug-fixing revisions. However, we have nothing to do with the source code. Instead, we use the bug ID which is mentioned in the revision log to query the corresponding bug report from the open bug repository of the software project. Secondly, both works use machine learning algorithms for classification, in particular Support Vector Machines (SVMs), While the features (in the machine learning sense), class labels and also the aim of the two approaches are completely different. Our goal is to predict the most related part of the software source file hierarchy to a newly reported bug. In contrast, they try to predict whether a particular change made by a developer to the source code is more likely to be buggy or clean. Further, we use textual information of bug reports in open bug repositories to form our features. However, they use properties of the change made to the software. An example of such property has been mentioned as the frequency of words that are present in the source code, before and after performing the change. Finally, our class labels are various revision paths in the software source file hierarchy, while their class labels are clean and buggy.

6 CONCLUSION & FUTURE WORK

In this paper, we have presented a new approach to localize bugs in the source file hierarchy of open source software projects. We have used Support Vector Machines (SVMs) for predicting the file path which is more likely to be related to a given software bug report, using its summary and description. The classifier has been trained using the information of fixed bugs in the past.

We have evaluated our approach on 'UI'component of the Eclipse Java Development Tool (JDT) project. Both precision and recall values are about 98%. Applying this approach on other FOSS projects remains as future work.

Removing stop-words and performing stemming are two common data preparation tasks in text categorization problems. Here, since the experimental results are satisfying even without such preparations, we decided not to get involved with them through this work. However, it is a worthy effort to examine the effects of those techniques on other FOSS projects in future work.

One part of our future work involves applying other machine learning algorithms to the same dataset and comparing the results. We are also interested in using our approach, in the field of automated bug triage, as discussed in Section 1.

Finally, one could extend the proposed approach in order to localize the bug, either in file level or on its exact line of code, instead of our hierarchical

directory level bug localization effort. Moreover, using our approach one could find the more buggy parts of the code in order to prioritize development tasks.

References

1. Deshpande A., Riehle D. (2008) The Total Growth of Open Source. The 4th International Conference on Open Source Systems (OSS 2008). Retrieved on November 27, 2009, from http://homepages.uc.edu/%7Edeshpaaa/oss-2008-total-growth-final.pdf.
2. Kroah-Hartman G., Corbet J., and McPherson A. (2008) Linux Kernel Development, How Fast it is Going. The Linux Foundation Publications. Retrieved on November 27, 2009, from https://www.linuxfoundation.org/publications/linuxkerneldevelopment.php.
3. Eclipse Bug Repository, https://bugs.eclipse.org/bugs, Verified on Nov. 24, 2009.
4. Anvik J., Hiew L., and Morphy G. C. (2006) Who Should Fix This Bug? In Proc. 28th International Conference on Software Engineering (ICSE 2006).
5. Anvik J., Morphy G. C. (2007) Determining Implementation Expertise from Bug Reports. 4th IEEE International Workshop on Mining Software Repositories (MSR 2007).
6. Baysal O., Godfrey M. W., and Cohen R. (2009) A Bug You Like: A Framework for Automated Assignment of Bugs. 17th IEEE International Conference on Program Comprehension (ICPC 2009).
7. Fogel K. (2005) Producing open source software. O'Reilly. First Edition, pp. 60-79.
8. Debian Bug Tracking System, http://www.debian.org/Bugs/, Verified on December 9, 2009.
9. Jeong G., Kim S., and Zimmermann T. (2009) Improving Bug Triage with Bug Tossing Graphs. The 7th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE).
10. Anatomy of Eclipse Bugs, Retrieved from http://www.bugzilla.org/docs/2.18/html/bug_page.html on December 19, 2009.
11. Life-cycle of Eclipse Bugs, Retrieved from http://www.bugzilla.org/docs/2.18/html/lifecycle.html on December 19, 2009.
12. Witten I. H., Frank E. (2005) Data Mining, Practical Machine Learning Tools & Techniques. Elsevier. Second Edition.
13. Bishop C. M. (2006) Pattern Recognition and Machine Learning. Springer, 2006.
14. Clarke E., Grumberg O., and Peled D. (1999) Model Checking. MIT Press.
15. Visser W., Havelund K., Brat G., and Park S. (2000) Model checking programs. In Proceeding of the 15th IEEE International Conference on Automated Software Engineering (ASE 2000).
16. Musuvathi M., Park D., Chou A., Engler D., and Cmc D. D. (2002) A pragmatic approach to model checking real code. In Proceeding of the 5th Symposium on Operating System Design and Implementation (OSDI 2002).
17. Liu C., Yan X., Fei L., Han J., Midkiff S. P. (2005) SOBER: Statistical Model-Based Bug Localization. The 3rd joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE).
18. Gyimothy T., Ferenc R., Siket I. (2005) Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. IEEE Trans. on Software Eng., vol. 31, no. 10, pp. 897-910, Oct. 2005.

19. Cleve H., Zeller A. (2005) Locating causes of program failures. In Proc. of 27th International Conference on Software Engineering (ICSE 2005).
20. Liblit B., Naik M., Zheng A., Aiken A., and Jordan M. (2005) Scalable statistical bug isolation. In Proc. of ACM SIGPLAN 2005 International Conference on Programming Language Design and Implementation (PLDI 2005).
21. Brun Y., Ernst M. D. (2004) Finding Latent Code Errors via Machine Learning over Program Executions. In Proc. of 26th International Conference on Software Engineering (ICSE 2004).
22. Ernst M. D., Perkins J. H., Guo P. J., McCamant S., Pacheco C., Tschantz M. S., and Xiao C. (2006) The Daikon System for Dynamic Detection of Likely Invariants. Science of Computer Programming, 2006.
23. Kim S., Whitehead Jr. E. J., and Zhang Y. (2008) Classifying Software Changes: Clean or Buggy? IEEE Trans. on Software Eng., vol. 34, no. 2, pp. 181-196, March/April 2008.
24. Shivaji S., Whitehead Jr. E. J., Akella R., and Kim S. (2009) Reducing Features to Imrove Bug Prediction. In Proceeding of the 15th IEEE International Conference on Automated Software Engineering (ASE 2009).
25. Salton G., Buckley C. (1988) Term-weighting approaches in automatic text retrieval. Information Processing & Management, vol. 24, Issue 5, pp. 513-523, 1988.
26. Plat J. C. (1998) Technical Report, MSR-TR-98-14, Microsoft Research, April 21, 1998
27. Plat J. C. (1998) Advances in Kernel Methods - Support Vector Learning. MIT Press, pp. 41-65, 1998.
28. WEKA 3-7-0 source comments, weka.classifiers.functions.SMO.
29. The official WEKA manual, Retrieved from http://www.cs.waikato.ac.nz/ml/weka/ on December 25, 2009.