Preemptive Bug Squashing

Defect Prediction Using Linux Kernel Patches

Chris J Arges
University of Texas
Austin,Texas,USA
christopherarges@gmail.com

ABSTRACT

Open source software presents a wealth of information; yet its interactions and processes can be complex or under-defined. Bugs present in this software introduce real costs for business and users. Thus, by using data mining techniques and rules, a method of proactively seeking bugs can be found. In order to achieve good results, data must be explored, gathered, pre-processed, modeled, and validated. This presents a wide array of challenges, and many opportunities for research. In this research proposal, a method of gathering and pre-processing data is explored. In addition, this data is mined to create a model that allows code to be input and classified as high-risk or low-risk for future defects. Thus, extra code reviews and quality checks can be applied to high-risk code patches to prevent bugs. This method is examined using the Linux kernel as a data set, and new and proposed patches as the input data. The goal being, if a project maintainer receives a patch, they can assess the risk of defect associated with the patch. Thus, the patch can be reviewed more closely and bugs can be avoided before being introduced into the code base.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

General Terms

Reliability, Design, Measurement

Keywords

Software Evolution, Metrics, Data Mining, Open Source

1. INTRODUCTION

Open source projects are no longer a niche community. These projects are used by businesses of all sizes around the world. While the software may be free, maintenance and quality assurance (QA) are a major cost when using these solutions in a business environment. To make the software better for everyone, bug reports can be filed against the offending project in hopes that those developers will fix the issue. Or the user can become a developer and submit a fix for their problem. Debugging introduces a very non-deterministic element into the software engineering process as debugging is difficult to plan. Some fixes may take seconds, while others take months to discover and fix. Therefore, reducing bugs is clearly an important goal.

1.1 Problem Description

Bugs are an inevitable; yet troublesome part of software development. In the open source world the software development process is distributed and varied. Any way to reduce the number of bugs can mean cost and time savings for the community and many businesses that rely on those open source projects. The problem is finding ways of reducing the numbers of bugs by making sense of the abundance of open data.

Information about bugs and source code is openly available to the public and presents a wealth of information. By using information taken from the source code, mailing lists, developer histories, revision control systems, social networking, and bug tracking tools, much information surrounding a project can be mined for data.

The basis of any Linux distribution is the Linux kernel and as such it is the focus of many bugs. Because Linux kernel development occurs semi-independently from the various distributions that use the project, the interactions between these project become complex. While other projects and interactions between distributions can be studied as well, for this project the scope is limited to Linux kernel bugs. Hopefully, these findings can be also used in other open source projects of a distributed nature and even traditional development type projects. To understand the problem space more completely, both Linux development and Linux distribution processes are described to explain how the software is developed, how bugs are discovered, and how fixes get introduced back into the project.

1.1.1 Linux Kernel Development Process

Linux kernel development isn't formally structured, but there are some implicit and explicit rules that most developers grasp once they start contributing. Explicit guidance is provided in the actual kernel documentation [1] and from various FAQs and books [8] [3]. The process is outlined very well at the following URL: `http://www.kernel.org/doc/Documentation/development-process/`. While the emphasis is on the Linux kernel, some similarities may exist in other open source projects. While it would be good to cover other projects as well, for this study the scope will be limited to just the Linux kernel.

The process begins with a developer downloading their own version of the Linux source tree. This can be accomplished using the 'git' code revisioning system using the command

'git clone', or downloading a source tarball. Then modifications are made to the kernel locally on the developer's machine. Normal testing occurs until the developer is satisfied with the change, then the developer signs off that code is acceptable to be distributed under the GPL. This change could be the addition of a new driver, support for a new filesystem, simple bug fixes, support for a new processor, or a simple spelling error in a kernel error message. The developer then determines which sub-system this patch affects and identifies the appropriate mailing list and developer to contact. This could be a particular subsystem like sound, or a particular architecture like powerpc. If it is a general patch, or needs to be seen by the larger community for discussion, it is sent to the LMKL (`lmkl.org`), and can be reviewed by the majority of the Linux developer community. After the maintainer and others on the mailing list review the patch, comments are sent back to the developer via email and the patch is updated. Once the maintainer is happy with the patch it may be included in the maintainers own tree. Eventually this patch gets pulled into Linus Torvalds's tree and becomes part of the 'mainline' kernel. [5] Not all patches are accepted, and larger, complex patches tend to be the most difficult to submit. [1]

When bugs are reported, they may be reported in a number of ways and through various outlets. Developers who work closely with maintainers may fix bugs and submit the fix via the mailing list without any formal bug report. Some bugs may be reported on a mailing list and may be taken care of by other developers. Bugs may be reported to the kernel bugzilla (`bugzilla.kernel.org`) which is an online bug tracking tool that can better track the progress of various fixes. To make matters even more complicated, bugs may also be reported to specific Linux distribution bug trackers. For instance Ubuntu, Debian, Red Hat, and SUSE all have their own tracking systems.

1.1.2 Linux Distribution Release Process

Overall, the open source landscape has many projects with complex dependencies. Most users need ways of navigating this sea of code, and as such there exist 'distributions' that package together useful programs to create an easy way to install a fully functional operating system on your computer. These Linux distributions include most programs that a typical user would expect from an operating system. Because of the open nature, there exist many different distributions to tailor a particular user's need or a business's need.

These distributions take in released versions of software from appropriate projects and package it together. In addition, they do a much higher degree of testing when these packages are integrated together. This testing involves selecting candidate versions of packages and proposing a plan for a release. Deadlines and milestones are set for the release to achieve the end goal of a distribution release. Testing occurs in parallel with packaging efforts and bugs are reported when found. These bugs are fixed either by developers in the distribution's community or from the actual project's community. Due to the asynchronous nature of releases between all projects, distributions need to freeze versions that they are using for a particular distribution release. [11] Fixes then are applied on top of these frozen versions and may or may not be sent back to the original project. When a release passes initial phases of testing, a release candidate is created which in turn is tested by a wider range of users. This varies depending on the distribution, but eventually after much testing an actual stable release is distributed.

2. EXISTING RESEARCH

There is some existing research; however the scope is somewhat different and emphasis is in different areas of the problem space.

Data mining of open source data has been explored by some research, although other studies are still needed. Li, Shaw and Herbsleb [7] did a study on how to predict field defects in OpenBSD which is another open source operating system. This is a good basis of study, and work to extend this to other projects such as Linux would be important. Menzies, Milton, Turhan, Cukic, Jiany and Bener [10] studied non-open source data-mining defect prediction using open data sets from NASA. In addition, Menzies [9] did another earlier along the same path.

Another challenge in this problem space is the reliability of bug reports. Because these reports are filled in by both volunteers and project developers, data can be missing or misleading. In addition, actually tracking the patch that fixed the code to the actual bug report is difficult to find. Thus, Bachmann, Bird, Rahman, Devanbu, and Bernstein [2] created a tool to facilitate linking of these bug reports in the Apache project to find the relevant code fix. Similarly, Zimmermann, Premraj, and Zeller [13] completed the difficult task of mapping bug defects to the source code locations in the open source Eclipse project. Thus the defects were treated almost as a metric of the code. In addition, other metrics were calculated and the data set was made public for others to test defect prediction models. Finally, Wu, Xie, Kaiser, and Passonneau [6] completed a project called BugMiner which was able to data mine an open source bug tracker like Bugzilla and predict future trends in bugs. This tool could be used to help auto complete new bug reports, thus increasing the quality of the reports.

Because of the vast amount of information available in any project, sorting through these factors manually can be error-prone and tedious. Another area of research covered by Nussbaum and Zacchiroli [12] was in creating a data-base for the Debian Linux distribution to store all relevant ongoing information thus enabling data mining. The target problem was enabling better QA (quality assurance), but could be used in a broader scope as well.

3. OBJECTIVES

The following are the main objectives of this research:

1. Collect relevant data, metrics, and bug reports about the Linux kernel (Data Discovery).

2. Process data to produce appropriate links and metrics (Data cleaning).

3. Produce a classification model that can use various metrics to predict if a patch will produce a bug or not (Model Development).

4. Validate results using hold-off data. (Model Assessment).

4. HYPOTHESIS

The following hypotheses will be experimentally tested during the course of the research:

1. Data from bug reports and source code repositories of the Linux kernel can be collected such that a link between the original code and the bug report can be found.

2. Using collected metrics, it is possible to analyze an incoming patch and predict with good accuracy if it is likely to introduce a defect or not. Good accuracy will be defined as producing a true positive and true negative rate that are both above 50% and in the case of existing research within a similar range that was found in the existing studies.

5. PROJECT PLAN

The project plan involves the standard steps in a data mining process. These steps will be considered the milestones in the research project. First data discovery will be done which involves both exploration and gathering. Next, preprocessing or cleaning of the data must be preformed. After this, model development occurs and a proper classification technique is selected. Input attributes are selected and refined to produce the optimal subset. Finally, the model is validated and assessed for accuracy until the results are satisfactory.

5.1 Data Discovery

Data discovery will occur on multiple fronts. This includes source code gathering, as well as collecting relevant bug reports.

First source code information must be collected, this most likely would entail collecting every single patch applied to the Linux kernel within a reasonable range of versions (1.0 to 3.0). This could be accomplished by checking out the kernel's git (source code managed) tree. The problem is that this would only cover a limited range of patches. In addition, patch sets would need to be downloaded from `kernel.org` until a sufficient number of patches and descriptions can be discovered. It would also be important to retain version information and temporal information in order to relate a particular patch or commit with a defect.

In addition to collecting the actual source code, it would be relevant to collect software metrics on the patches that have been collected. Thus, for each patch a set of metrics such as code complexity, SLOC, maintainability, coupling, and others would be collected. In addition to these metrics other factors such as the number of files the patch modifies, or the number of lines of code the patch modifies will also be useful. This data would then be stored in a format that would make it easy to retrieve later such as a database or a spreadsheet.

Next, bug report information must be collected from multiple sites possibly including: `https://bugzilla.kernel.org/`, `https://bugs.launchpad.net/~kernel-bugs`, `https://bugzilla.redhat.com/`, `http://www.debian.org/Bugs/`. These bug reports would need to be gathered in a regular way as to have a canonical format that can be easily processed. Subsequent fixes to the bugs would need to be discovered and information about the fix would need to be determined. If a particular commit was cited as fixing the bug, this would be critical to collect. This information would be stored into a database or spreadsheet such that it can be retrieved later.

patch#1	metric1	metric2	...	metricN	BUG
patch#2	metric1	metric2	...	metricN	OK

Table 1: Example Data

5.2 Data Cleaning

While other studies [2] have been done in linking bug reports to fixes, this research would be novel in that the dataset is the Linux kernel. The data gathered for the bug reports would need to only consider bugs that have a valid resolution. In addition, this resolution would need to have been the result of a code change that could be ascertained from the bug information its-self. While this may limit the amount of bugs that can be mined, it is hoped that with the abundance of bug trackers plenty of valid data should be able to be used. Thus some filtering of the bug data must be preformed to ensure we have acceptable data to work with. The static code metrics would not need to be filtered given we will be calculating this data.

An example data set after cleaning would could look similar to what is shown in Table 1.

5.3 Model Development

Once the data is in order then a model can be developed to predict defects in code patches. This part of the project cannot be fully defined as some trial and error will be required to select the base technique and combination of metrics for the best performance. However, an outline of required steps can be explained.

First, metrics should be collected and explored. This means determining if there are correlations between metrics or visualizing the data to find patterns. By using this insight, one can make better decisions for mining.

Next, techniques such as Linear Discriminant Analysis (LDA) or Principle Component Analysis (PCA) are used to determine the best subset of features that maximize variance. This will also be iterative as some parameters might be found to be helpful or harmful to the model's accuracy.

After parameters are chosen a classification algorithm is chosen. The output will be two classes initially: BUG or OK. Decision trees, random forests, or even Bayesian approaches could be used as our classifier. This would need to be determined by trail and error and further understanding of the data.

	Actual Bug	Actually OK
Predicted Bug	True Positive (TP)	False Positive (FP)
Predicted OK	False Negative (FN)	True Negative (TN)

Table 2: Confusion Matrix Example [4]

5.4 Model Assessment

The model assessment and model development may occur multiple times as the model may need to be refined to achieve the desired outcome. The model will be validated against existing data by partitioning the data into training and validation data. If the entire data set is used, the model may fit too well and not be well adapted to correctly predicting new data points. The performance of the classifier will be assessed by examining the True Positive (TP) and True Negative (TN) values of the binomial classifier. Table 2 shows an example of the confusion matrix which illustrates what the TP and TN values mean.

This table shows that a TP is defined as the model predicting a bug and the data showing the input patch caused an actual bug from the data set. Likewise a TN is defined as an input patch that did not cause a bug being predicted as OK (no bug). These values will vary from 0 to 1, or from %0 to %100. A TP of %100 and TN of %100 would be the most desirable model performance.

Menzies was able to achieve a TP value of %71, and a TN of %75 in the overall study [9]. So the hope is that this research yields a similar performance. But due to the differences in the data set (one being closed-source and the other open), there may be major differences in these values.

6. CONCLUSION

By using the large magnitude of data available in an open source project, a system of identifying code patches that have a high risk of defects can be predicted. These predictions would allow maintainers to make better decisions about the code that is being introduced into their code base. By looking at the Linux kernel in particular, this research has a large amount of data in terms of source code and in terms of bug reports. Aggregating these bug reports and linking them to the original areas of source code will help identify which patches caused which bugs. Metrics will be calculated based on the code in the patch and the areas it affects in the Linux kernel. This data will be cleaned and explored to find the best subset of parameters to use in the prediction model. A binomial classifier will be trained with a training partition of the data. This data will be validated with a validation partition of the data. Once the accuracy is acceptable the model will be documented and studied further.

7. REFERENCES

[1] Linux kernel documentation. `http://www.kernel.org/doc/Documentation`.

[2] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: Bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 97–106. ACM, 2010.

[3] D. Bovet, M. Cesati, and A. Oram. *Understanding the Linux kernel*. O'Reilly & Associates, Inc., 2002.

[4] J. Ghosh. Data mining - class note. Online, 2010.

[5] G. Hertel, S. Niedner, and S. Herrmann. Motivation of software developers in open source projects: an internet-based survey of contributors to the linux kernel. *Research policy*, 32(7):1159–1177, 2003.

[6] L. Kaiser and R. Passonneau. Bugminer: Software reliability analysis via data mining of bug reports. *delta*, 12(10):09–0500.

[7] P. Li, J. Herbsleb, and M. Shaw. Finding predictors of field defects for open source software systems in commonly available data sources: A case study of openbsd. In *Software Metrics, 2005. 11th IEEE International Symposium*, pages 10–pp. IEEE, 2005.

[8] R. Love. *Linux kernel development*. Addison-Wesley Professional, 2009.

[9] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, pages 2–13, 2007.

[10] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[11] M. Michlmayr, F. Hunt, and D. Probert. Release management in free software projects: Practices and problems. *Open Source Development, Adoption and Innovation*, pages 295–300, 2007.

[12] L. Nussbaum and S. Zacchiroli. The ultimate debian database: Consolidating bazaar metadata for quality assurance and data mining. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 52–61. IEEE.

[13] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 9. IEEE Computer Society, 2007.