

# Software Defect Prediction: Review, Commentary, and Future Work

Matthew Neal  
Department of Computer  
Science  
North Carolina State  
University  
Raleigh, North Carolina, USA  
meneal@ncsu.edu

Joseph Sankar  
Department of Computer  
Science  
North Carolina State  
University  
Raleigh, North Carolina, USA  
jesankar@ncsu.edu

Alexander Sobran  
Department of Computer  
Science  
North Carolina State  
University  
Raleigh, North Carolina, USA  
aisobran@ncsu.edu

## ABSTRACT

This paper provides a comprehensive review and commentary on software defect prediction research. Review and criticism of previous work is presented. Future steps in the domain are given.

## Keywords

Fault prediction model, Software mining, Ant Colony Optimization, Classification, Defect Prediction

## 1. INTRODUCTION

It is a well-known fact that software bugs are much cheaper and easier to fix before being released. But finding these bugs is often difficult and may not be cost effective to fix. Researchers have been working on fault detection models to predict which software modules are most likely to contain bugs post-release. Management can use these predictions to focus testing and bug-fixing efforts on those modules, resulting in fewer bugs in release which are less costly to fix.

In this paper, we focus on the advances made in software fault prediction models in the literature. Section 2 contains an assortment of existing work in the area.

## 2. RELATED WORK

Cagatay Catal and Banu Diri [3] gave an overview of software fault prediction studies and advancements up to 2008. They found that an increasing number of studies used datasets available to the public. They also found that since 2005, machine learning algorithms have become increasingly popular choices to implement the models. Finally, they observed that the most dominant metrics in fault prediction were at the method level. They recommended that machine learning algorithms and public datasets continue to be used, but caution against using method-level metrics and instead

suggested class-level metrics as they can predict faults earlier in the software development cycle.

Vandecruys et al [9] mined software repositories to create predictive models. The authors used AntMiner+, an Ant Colony Optimization (ACO)-based classification technique. On public datasets, AntMiner+ was found to be competitive to alternative classification techniques, such as C4.5, logistic regression, and support vector machines. The authors suggested that software managers would find the output of rules produced by AntMiner+ easy to understand and accept.

While there have been plenty of studies about predicting software faults based on the code itself, few studies have looked at organizational structure as a factor. Nagappan et al [5] used organizational metrics, such as number of engineers, edit frequency, and organizational intersection factor to predict fault-proneness. The authors compared the effectiveness of the resulting model with alternative models which use traditional software metrics, like code churn and code complexity. They found that the model derived from organizational metrics had better precision and recall than others derived from software metrics, meaning they can also be effective indicators of failure-proneness.

Bird et al [2] examined whether development that is largely distributed produces more failures than development that is mainly collocated. The belief at the time was that global development was prone to more failures than collocated development. They found a negligible difference in both code metrics and failures between the two methods of development. The authors examined how the developers working on Windows Vista managed to work well together among teams located in different countries based on the relationships between the development sites, cultural barriers, communication, consistent use of tools, end to end ownership, common schedules, and organizational integration. They recommended that companies wishing to distribute development across sites located far apart to employ similar strategies to overcome some of the difficulties associated with such an endeavor.

Arisholm et al [1] examined different ways to build and evaluate fault prediction models. First, they tested a variety of modeling techniques, such as neural networks, C4.5 with some variants, support vector machines, and logistic regression. They then looked at different metrics: Process measures, object oriented code measures, and delta measures. Finally, they looked at ROC area and cost-effectiveness as evaluation criteria. They found that the choice of model-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

WOODSTOCK '97 El Paso, Texas USA

© 2015 ACM. ISBN 123-4567-24-567/08/06...\$15.00

DOI: 10.475/123.4

ing technique did not have much of an impact when evaluated with both criteria tested. Among the metric sets, they found that process measures provide a significant improvement over the others, even though they are typically more costly to collect. They also suggested cost-effectiveness as a good evaluation technique instead of traditional techniques like precision and recall as smarter decisions can be made to prioritize which parts of the project are tested and bug-fixed.

Jun Zheng [10] argued that the mistakenly predicting a module as non-defective is more dangerous and costly than mistakenly predicting a module as defective. He presented three algorithms which boost neural networks to predict software defects with cost in mind. When evaluated on four NASA datasets, he found that threshold moving algorithm gave the best results in the sole Normalized Expected Cost of Misclassification (NECM) measure. He especially recommended threshold-moving algorithms on projects written in object oriented languages.

Most software products are structured in a hierarchical format, for example into methods, classes, files, packages, etc. What level of analysis should fault prediction models operate on? And can analysis on one level of study apply to other levels? Posnett et al [7] examines these issues. They observed that sometimes relevant phenomena only occur at an aggregated level or data may only be observed at an aggregated level, meaning that studies are often conducted at aggregated levels. They also noted that in other fields of study there are ecological fallacies which make findings at aggregated levels not apply at disaggregated levels. They found that much of these fallacies also exist in the domain of computer software and that care needs to be taken when employing ecological inference. As to how exactly the risks of ecological fallacies can be dealt with, the authors left open for future research.

Traditional software fault prediction models are trained on historical project data. Since new projects do not have such a volume of historical data, these fault prediction models are not as effective at within-project defect prediction. A possible solution to this issue is cross-project prediction which uses data from one project to predict defects in another. Still, models using this technique exhibit poor performance. Rahman et al [8] argued that one reason for such behavior is that standard evaluation measures such as precision, recall, and F-measure are taken at specific threshold settings, while they really should be taken in a range of time/cost vs. quality trade offs. They took the standard measures at a variety of tradeoffs and found that cross-project defect prediction is at least as good as within-project defect prediction, and sometimes substantially better.

Also within the area of cross-project defect prediction, Nam et al [6] found that when the source and target projects have a different feature distributions, the resulting model gives poor performance. The authors use Transfer Component Analysis (TCA) to find a common feature set for both projects and then map the data of both projects to it. They found that TCA is sensitive to normalization, so they developed an improvement, TCA+ to select appropriate normalization options. After using TCA+ to create cross-project defect prediction models for eight open-source projects, they found a significant improvement in prediction performance compared to traditional algorithms and techniques. The authors proposed applying knowledge in one domain to another to further improve performance.

As another approach to improving fault prediction, Tian Jiang [4] introduced *personalized defect prediction*. He argued that developers have different coding styles and techniques and that if each developer had their own defect prediction model, performance would improve. He was careful to note that the developer was not a feature of the model, but that there was a model for each developer. He ran experiments on six large open-source projects using PCC+, a model which chooses the highest confidence prediction among CC (traditional change classification), PCC (personalized change classification), and weighted PCC (PCC with changes from other developers added to a developer's model). Compared to CC and MARS (another predictor which also creates different models for different groups of data), PCC+ outperformed CC, MARS, and PCC as long as there is enough training data for each developer. Jiang recommended that PCC+ be applied to other recommendation systems and types of predictions.

### 3. MACHINE LEARNING ALGORITHMS

### 4. METRICS

### 5. FEATURES

### 6. DATA SOURCES

The choice of dataset is extremely important, especially when doing software defect prediction research. Datasets should be representative of actual software projects and should ideally be accessible to the public so the results can be re-tested and verified by others. Within the referenced papers, we found a variety of data sources which we feel need to be discussed. Primarily, we found open-source datasets, NASA datasets, and Windows Vista datasets. We will discuss each along with some reasons why they may have been chosen.

#### 6.1 Open-source

#### 6.2 NASA

A few papers (Zheng [10]...) used datasets from NASA. Among them were KC1, used for storage management, KC2, used for scientific data processing, CM1, used for NASA spacecraft instruments, and PC1, used for flight software.

One major reason for using NASA datasets is that most of their software is critical. A malfunction or crash due to a software defect is not only costly, but can also be deadly. NASA's software needs to be thoroughly tested and bug-fixed before it is ready for use. Software defect models which work well on these datasets can be claimed to be more reliable and trustworthy. Additionally, NASA's datasets contain code metrics and defect information, which is easy to mine and create training and test data from.

NASA datasets are publicly available. Researchers can more easily verify the results obtained in a study. Also, since these datasets are static, they can be used as a baseline to compare the results from a newly created model against those from other studies.

#### 6.3 Windows Vista

Two of the referenced papers (Bird et al [2] and Nagappan et al [5]) use data from Windows Vista. Neither paper

tests a fault prediction model using Vista test data. Instead, the first paper examined how different development teams worked together within Microsoft and the second paper examined Microsoft’s organizational structure. Two of the three authors of the second paper are associated with Microsoft Research.

But why was Windows Vista chosen? Windows in general is a widely used operating system which contains many lines of code. Windows is also known for being vulnerable without frequent patching and antivirus software. Windows Vista may have been the latest version of Windows at the time of both papers, but Vista is also known as a slow and buggy release. Maybe the authors intended to show that Vista did not contain an unusually high number of bugs compared to other releases, or maybe they just wanted to choose a Microsoft product that was well-known, whether for good or bad.

## 7. DATA PREPROCESSING

## 8. CROSS-PROJECT DEFECT PREDICTION

### 8.1 Context Factors

**Context factors** for projects were assessed by Fend et al (2014 cite here). Used as a measurement of project similarity. The provides a set of measurements to rank a project’s relatedness when performing cross-project defect prediction.

**Programming Languages** Projects can be divided into groups depending on the programming language they are written in. Projects written in multiple languages could be problematic unless multiple language categories are included.

**Issue Tracking** Whether or not a project uses an issue tracking system.

**Total Lines Of Code** Project size based on the total number of lines of code split by quartiles.

**Total Number of Commits** Project size as a function of the number of commits. Split into quartiles.

**Total Number of Developers** Project size as a determined by the number of developer. Split into quartiles.

### 8.2 Ranking

### 8.3 Clustering

### 8.4 Math Equations

You may want to display math equations in three distinct styles: inline, numbered or non-numbered display. Each of the three are discussed in the next sections.

#### 8.4.1 Inline (In-text) Equations

A formula that appears in the running text is called an inline or in-text formula. It is produced by the **math** environment, which can be invoked with the usual `\begin. . . \end` construction or with the short form `$ . . . $`. You can use any of the symbols and structures, from  $\alpha$  to  $\omega$ , available in L<sup>A</sup>T<sub>E</sub>X[?]; this section will simply show a few examples of in-text equations in context. Notice how this equation:  $\lim_{n \rightarrow \infty} x = 0$ , set here in in-line math style, looks slightly different when set in display style. (See next section).

#### 8.4.2 Display Equations

A numbered display equation – one set off by vertical space from the text and centered horizontally – is produced by the **equation** environment. An unnumbered display equation is produced by the **displaymath** environment.

Again, in either environment, you can use any of the symbols and structures available in L<sup>A</sup>T<sub>E</sub>X; this section will just give a couple of examples of display equations in context. First, consider the equation, shown as an inline equation above:

$$\lim_{n \rightarrow \infty} x = 0 \tag{1}$$

Notice how it is formatted somewhat differently in the **displaymath** environment. Now, we’ll enter an unnumbered equation:

$$\sum_{i=0}^{\infty} x + 1$$

and follow it with another numbered equation:

$$\sum_{i=0}^{\infty} x_i = \int_0^{\pi+2} f \tag{2}$$

just to demonstrate L<sup>A</sup>T<sub>E</sub>X’s able handling of numbering.

### 8.5 Citations

Citations to articles [?, ?, ?, ?], conference proceedings [?] or books [?, ?] listed in the Bibliography section of your article will occur throughout the text of your article. You should use BibTeX to automatically produce this bibliography; you simply need to insert one of several citation commands with a key of the item cited in the proper location in the .tex file [?]. The key is a short reference you invent to uniquely identify each work; in this sample document, the key is the first author’s surname and a word from the title. This identifying key is included with each item in the .bib file for your article.

The details of the construction of the .bib file are beyond the scope of this sample document, but more information can be found in the *Author’s Guide*, and exhaustive details in the *L<sup>A</sup>T<sub>E</sub>X User’s Guide*[?].

This article shows only the plainest form of the citation command, using `\cite`. This is what is stipulated in the SIGS style specifications. No other citation format is endorsed or supported.

### 8.6 Tables

Because tables cannot be split across pages, the best placement for them is typically the top of the page nearest their initial cite. To ensure this proper “floating” placement of tables, use the environment **table** to enclose the table’s contents and the table caption. The contents of the table itself must go in the **tabular** environment, to be aligned properly in rows and columns, with the desired horizontal and vertical rules. Again, detailed instructions on **tabular** material is found in the *L<sup>A</sup>T<sub>E</sub>X User’s Guide*.

Immediately following this sentence is the point at which Table 1 is included in the input file; compare the placement of the table here with the table in the printed dvi output of this document.

To set a wider table, which takes up the whole width of the page’s live area, use the environment **table\*** to enclose the table’s contents and the table caption. As with a

Table 1: Frequency of Special Characters

Non-English or Math	Frequency	Comments
Ø	1 in 1,000	For Swedish names
$\pi$	1 in 5	Common in math
\$	4 in 5	Used in business
$\Psi_1^2$	1 in 40,000	Unexplained usage

Figure 1: A sample black and white graphic.

single-column table, this wide table will “float” to a location deemed more desirable. Immediately following this sentence is the point at which Table 2 is included in the input file; again, it is instructive to compare the placement of the table here with the table in the printed dvi output of this document.

## 8.7 Figures

Like tables, figures cannot be split across pages; the best placement for them is typically the top or the bottom of the page nearest their initial cite. To ensure this proper “floating” placement of figures, use the environment **figure** to enclose the figure and its caption.

This sample document contains examples of **.eps** files to be displayable with L<sup>A</sup>T<sub>E</sub>X. If you work with pdfL<sup>A</sup>T<sub>E</sub>X, use files in the **.pdf** format. Note that most modern T<sub>E</sub>X system will convert **.eps** to **.pdf** for you on the fly. More details on each of these is found in the *Author’s Guide*.

As was the case with tables, you may want a figure that spans two columns. To do this, and still to ensure proper “floating” placement of tables, use the environment **figure\*** to enclose the figure and its caption. and don’t forget to end the environment with **figure\***, not **figure**!

## 8.8 Theorem-like Constructs

Other common constructs that may occur in your article are the forms for logical constructs like theorems, axioms, corollaries and proofs. There are two forms, one produced by the command **\newtheorem** and the other by the command **\newdef**; perhaps the clearest and easiest way to distinguish them is to compare the two in the output of this sample document:

This uses the **theorem** environment, created by the **\newtheorem** command:

THEOREM 1. *Let  $f$  be continuous on  $[a, b]$ . If  $G$  is an antiderivative for  $f$  on  $[a, b]$ , then*

$$\int_a^b f(t)dt = G(b) - G(a).$$

The other uses the **definition** environment, created by the **\newdef** command:

Definition 1. If  $z$  is irrational, then by  $e^z$  we mean the unique number which has logarithm  $z$ :

$$\log e^z = z$$

Figure 2: A sample black and white graphic that has been resized with the **includegraphics** command.

Figure 4: A sample black and white graphic that has been resized with the **includegraphics** command.

Two lists of constructs that use one of these forms is given in the *Author’s Guidelines*.

There is one other similar construct environment, which is already set up for you; i.e. you must *not* use a **\newdef** command to create it: the **proof** environment. Here is a example of its use:

PROOF. Suppose on the contrary there exists a real number  $L$  such that

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = L.$$

Then

$$l = \lim_{x \rightarrow c} f(x) = \lim_{x \rightarrow c} \left[ gx \cdot \frac{f(x)}{g(x)} \right] = \lim_{x \rightarrow c} g(x) \cdot \lim_{x \rightarrow c} \frac{f(x)}{g(x)} = 0 \cdot L = 0,$$

which contradicts our assumption that  $l \neq 0$ .  $\square$

Complete rules about using these environments and using the two different creation commands are in the *Author’s Guide*; please consult it for more detailed instructions. If you need to use another construct, not listed therein, which you want to have the same formatting as the Theorem or the Definition shown above, use the **\newtheorem** or the **\newdef** command, respectively, to create it.

## A Caveat for the T<sub>E</sub>X Expert

Because you have just been given permission to use the **\newdef** command to create a new form, you might think you can use T<sub>E</sub>X’s **\def** to create a new command: *Please refrain from doing this!* Remember that your L<sup>A</sup>T<sub>E</sub>X source code is primarily intended to create camera-ready copy, but may be converted to other forms – e.g. HTML. If you inadvertently omit some or all of the **\defs** recompilation will be, to say the least, problematic.

## 9. CONCLUSIONS

This paragraph will end the body of this sample document. Remember that you might still have Acknowledgments or Appendices; brief samples of these follow. There is still the Bibliography to deal with; and we will make a disclaimer about that here: with the exception of the reference to the L<sup>A</sup>T<sub>E</sub>X book, the citations in this paper are to articles which have nothing to do with the present subject and are used as examples only.

## 10. ACKNOWLEDGMENTS

This section is optional; it is a location for you to acknowledge grants, funding, editing assistance and what have you. In the present case, for example, the authors would like to thank Gerald Murray of ACM for his help in codifying this *Author’s Guide* and the **.cls** and **.tex** files that it describes.

## 11. REFERENCES

- [1] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *J. Syst. Softw.*, 83(1):2–17, Jan. 2010.

**Table 2: Some Typical Commands**

Command	A Number	Comments
<code>\alignauthor</code>	100	Author alignment
<code>\numberofauthors</code>	200	Author enumeration
<code>\table</code>	300	For tables
<code>\table*</code>	400	For wider tables

**Figure 3: A sample black and white graphic that needs to span two columns of text.**

- [2] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy. Does distributed development affect software quality?: An empirical case study of windows vista. *Commun. ACM*, 52(8):85–93, Aug. 2009.
- [3] C. Catal and B. Diri. A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4):7346 – 7354, 2009.
- [4] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 279–289, Nov 2013.
- [5] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: An empirical case study. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, pages 521–530, New York, NY, USA, 2008. ACM.
- [6] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 382–391, Piscataway, NJ, USA, 2013. IEEE Press.
- [7] D. Posnett, V. Filkov, and P. Devanbu. Ecological inference in empirical software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11*, pages 362–371, Washington, DC, USA, 2011. IEEE Computer Society.
- [8] F. Rahman, D. Posnett, and P. Devanbu. Recalling the “imprecision” of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12*, pages 61:1–61:11, New York, NY, USA, 2012. ACM.
- [9] O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer, and R. Haesen. Mining software repositories for comprehensible software fault prediction models. *J. Syst. Softw.*, 81(5):823–839, May 2008.
- [10] J. Zheng. Cost-sensitive boosting neural networks for software defect prediction. *Expert Systems with Applications*, 37(6):4537–4543, 2010.

## APPENDIX

### A. HEADINGS IN APPENDICES

The rules about hierarchical headings discussed above for the body of the article are different in the appendices. In the **appendix** environment, the command **section** is used to indicate the start of each Appendix, with alphabetic order designation (i.e. the first is A, the second B, etc.) and a title (if you include one). So, if you need hierarchical structure

within an Appendix, start with **subsection** as the highest level. Here is an outline of the body of this document in Appendix-appropriate form:

#### A.1 Introduction

#### A.2 The Body of the Paper

##### A.2.1 Type Changes and Special Characters

##### A.2.2 Math Equations

*Inline (In-text) Equations.*

*Display Equations.*

##### A.2.3 Citations

##### A.2.4 Tables

##### A.2.5 Figures

##### A.2.6 Theorem-like Constructs

*A Caveat for the T<sub>E</sub>X Expert*

### A.3 Conclusions

### A.4 Acknowledgments

### A.5 Additional Authors

This section is inserted by L<sup>A</sup>T<sub>E</sub>X; you do not insert it. You just add the names and information in the `\addition-alauthors` command at the start of the document.

### A.6 References

Generated by bibtex from your .bib file. Run latex, then bibtex, then latex twice (to resolve references) to create the .bbl file. Insert that .bbl file into the .tex source file and comment out the command `\thebibliography`.

## B. MORE HELP FOR THE HARDY

The sig-alternate.cls file itself is chock-full of succinct and helpful comments. If you consider yourself a moderately experienced to expert user of L<sup>A</sup>T<sub>E</sub>X, you may find reading it useful but please remember not to change it.