An Objective Measure of Code Quality

Mark Dixon
CTO, Enerjy Software
mark_dixon@enerjy.com

Abstract

1. Introduction

The work described in this paper attempts to solve two problems. First, meeting the need for a simple, single metric of source code quality. While any single metric must inherently omit many aspects of code quality, this is offset by the clarity and simplicity it brings. This is well illustrated by Edward Tufte [Tufte97], who explores the difficulty engineers experienced trying to convince management that it was unsafe to launch the space shuttle Challenger in freezing temperatures. There was existing evidence that the rubber o-rings in the solid-fuel boosters experienced damage at lower launch temperatures, but the damage was classified into four different categories. This obscured the relationship between damage and temperature. By combining the damage into a single 'damage index' and plotting it against temperature, Tufte clearly highlights the riskiness of the launch.

Second, there is very little hard evidence of the benefits of source code metrics, (such as size and complexity) and static analysis. While many organizations have coding standards, those standards are often somewhat arbitrary and often fall into disuse. Proponents of the standards have no specific arguments to justify the perceived overhead they impose on the development process.

The approach we take is to perform a historical analysis of a large body of source code to determine a statistical relationship between certain source code metrics and code quality. With this analysis in place, we can then use the statistical model to assign a quality score to any source file.

2. Code Quality

The first task is to define what we mean by code quality. Here we follow the example of Denaro and Pezze [Denaro02] and base our definition on the concept of *fault-proneness*. For most organizations, the ultimate requirement for a source file is that it contains code that functions correctly. While there are other desirable characteristics, in particular minimizing cost of maintenance, correctness is generally the primary driver. There is also very little data available on the maintenance cost of individual source files, making it very hard to perform any analysis. Most projects, however, use a source code control system that describes the reason for every code change. This makes it straightforward to identify which files contained faults requiring a code change to fix.

A *fault-prone* file is one that contains a disproportionate number of faults. Specifically, for each file, determine how many faults were fixed in that file over a given time period. After ranking the files in descending order of the number of faults, the fault-prone files are the files at the top of the list that together account for a predetermined proportion of the total number of faults.

Assuming that we have an algorithm to determine the probability that a source file is fault-prone, we can now define a code quality score using the simple formula

*Score = 10 * (1 - Probability(file is fault-prone))*

The score is scaled to run from 0 to 10, with files that have a very high likelihood of being fault-prone scoring near 0 and files that are very unlikely to be fault-prone scoring near 10.

Given a quality score for a file, we then define the score for a package or project to be the mean (i.e., average) of all of the contained files. In practice, the score for a file is usually 0 or 10 and rarely falls in between. Thus the score for a project can be thought of as representing the proportion of fault-prone files within that project.

The remainder of this paper describes the process for predicting the probability that a given file is fault-prone.

3. Training Data

Classifying a collection of objects into categories based on their attributes is a common problem in data mining. A typical example is a spam filter that attempts to classify documents into spam and non-spam based on the content of the documents. In our case, we need to classify source files into fault-prone and non-fault-prone based on the values of a number of source code metrics. If we can construct such a classifier then we have two benefits. First, most classifiers actually predict a probability that a file is fault-prone rather than an absolute yes/no answer. That probability is exactly what we need for the quality score. Second, the classifier will identify which metrics are effective predictors of fault-proneness.

Any classifier requires a body of training data. We collected the complete history of 12 popular, open-source Java projects. The projects are:

Project	Description
Azureus	Java BitTorrent Client
Commons Collections	Builds on the JDK Collections Framework by providing new interfaces, implementations and utilities

Project	Description
Commons Digester	A configurable XML -> Java mapping module
Commons Logging	An ultra-thin bridge between different logging implementations
Cruisecontrol	A framework for a continuous build process
ehcache	A widely used Java distributed cache for general purpose caching, Java EE and light-weight containers
FindBugs	A program which uses static analysis to look for bugs in Java code
Jakarta ORO	A set of text-processing classes that provide Perl5 compatible regular expressions
Jakarta Regexp	A 100% Pure Java Regular Expression package
Apache Lucene	A high-performance, full-featured text search engine library written entirely in Java
Spring Framework	The leading full-stack Java/JEE application framework
Apache Velocity	A free open-source templating engine

For each project, we identified faults by searching the source code control system's history for check-in comments containing the words *bug* or *fix*. A manual check on a sample of the projects showed that, while this very crude approach did tend to over-count faults, the error was less than 5%. For each check-in that fixed a fault, we incremented the fault count by 1 for every file that was changed in that check-in. The final data set contained 3817 files, of which 420 (11%) were classified as fault-prone.

Additionally, for each file we collected a total of 228 source metrics. 33 metrics were general source metrics such as the size of the source file, the number of lines of code and classic McCabe and Halstead complexity measures. The remaining 195 were the number of violations recorded for each of the coding standards defined by the Enerjy Code Analyzer. Very similar results would be achieved using a different analyzer such as Checkstyle, PMD or FindBugs.

4. Classification Model

There are several approaches to the classification problem. For a good overview, see Witten and Frank's *Data Mining* [Witten05] or, for a more rigorous approach, *The Elements of Statistical Learning* by Hastie et al [Hastie01]. Denaro and Pezze successfully used a logistic regression model to predict fault-proneness based on a

selection of up to five of the source metrics. We were not able to replicate their success with our training data. Instead we used a naive Bayesian model.

The general approach behind a naive Bayesian model is to assume that all of the metrics are independent, and model each metric separately for fault-prone files and non-fault-prone files. Bayes theorem then provides a formula to combine the information from each metric into an overall probability that a file is fault-prone.

To look at a specific example, consider the SIZE metric, which is simply the number of characters in the source file. We chose to model all source metrics using a Normal distribution and all Analyzer violation metrics using a Poisson distribution. For our training data, we found that the SIZE metric had an average value of 14,461 characters in fault-prone files but only 4,074 in non-fault-prone files. Figure 1 shows the probability distributions for both types of file.

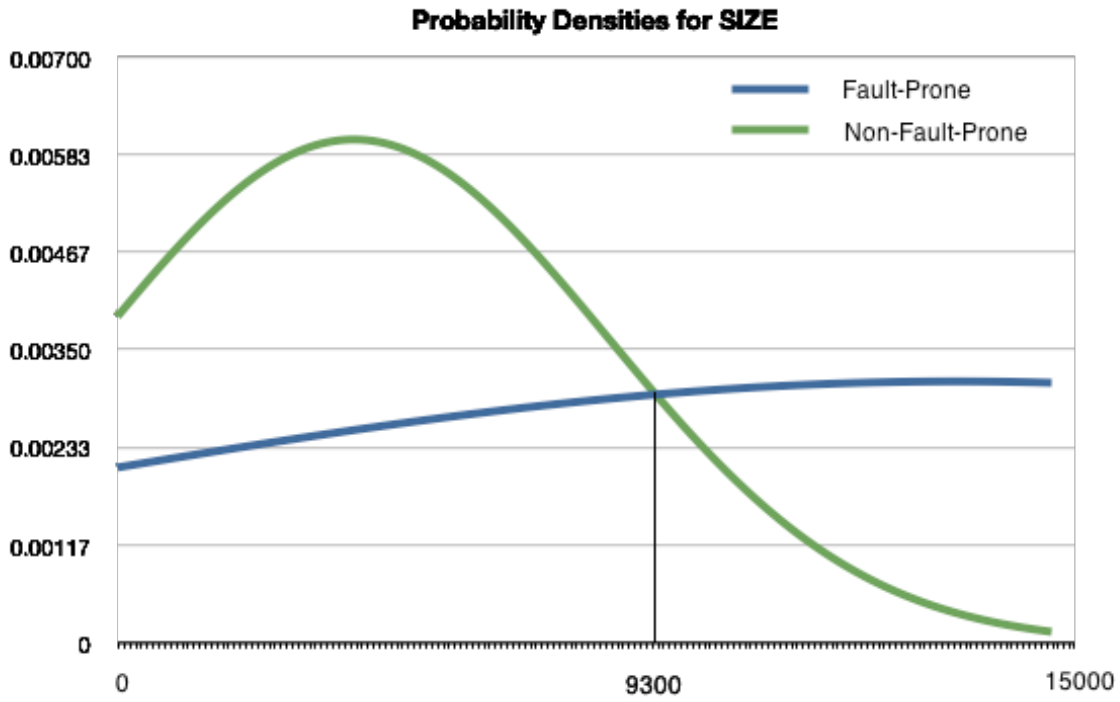

Figure 1

Intuitively, this chart shows that small files are more likely to be non-fault-prone. This continues until the file size reaches around 9300 characters, at which point it becomes more likely that the file is fault-prone. Bayes Theorem provides a way to formalize this intuition, and additionally to combine the results for multiple metrics.

5. Results

The primary result is that we were able to generate a model that was an effective predictor of fault-proneness. For 11 of the 12 projects, the model predicted fault-proneness with a classification error rate of around 15%. For the remaining project (Velocity) the error rate was around 25%.

Second, the assumptions behind the Bayesian model were tested using a Lilliefors test for the normally distributed metrics and a standard chi-squared test for the poisson distributed metrics. The distributions were found to be a reasonable fit at a 95% confidence level for many of the metrics.

Among the source metrics, the most effective predictors were:

Metric	Description
PROGRAM_VOLUME	Halstead program volume
EXEC_COMMENTS	Comments in executable code
LINE_COMMENTS	Number of line comments

Among the analyzer metrics, the most effective predictors were:

Metric	Description
JAVA0034	Missing braces in *if* statement
JAVA0117	Missing javadoc for method
JAVA0110	Incorrect javadoc: no @return tag

In all cases, larger values of the metrics indicate fault-proneness.

Some of the analyzer metrics were not useful predictors simply because they did not occur in the training data. A richer set of training data should lead to a better model. We will continue to refine the model over time, and will continue to publish our results in this format.

We ran the model on a number of open-source projects and the results generally matched our expectations, with projects known for their quality scoring high, and others scoring lower. Full results are available at the Enerjy Index site (http://www.enerjy.com)

6. Future Work

There are a number of future directions for this work:

- Better understand why the model does not fit Velocity well. Although we believe the current model to be a good, generalized predictor of fault-proneness, we may be able to improve its accuracy by developing a family of models for different types of application.
- Add more projects to the training data. This may bring more of the Analyzer violations into play as useful detectors of fault-proneness.
- The current model uses absolute metrics, which are all somewhat influenced by the file's size. Build a model that uses metrics scaled by the file size (i.e. number of violations per line of code rather than just number of violations). Our early results with these models have not been promising, but they have the potential to bring more independent data to the analysis.

Finally, it is our intention to make our training data generally available. If you would be interested in using it, please contact us using the address at the top of this paper.

References

[Denaro02] G. Denaro and M. Pezze. An Empirical Evaluation of Fault-Proneness Models. *Proc. International Conf on Software Engineering (ICSE2002)*, Miami, USA, May 2002

[Hastie01] Trevor Hastie, Robert Tibshirani and Jerome Friedman. *The Elements of Statistical Learning*. Springer, 2001.

[Tufte97] Edward R Tufte, *Visual Explanations* p38-53. Graphics Press LLC, 1997.

[Witten05] Ian H Witten and Eibe Frank. *Data Mining - Practical Machine Learning Tools and Techniques*. Morgan Kaufman, 2005