INTERNATIONAL JOURNAL OF INTELLIGENT CONTROL AND SYSTEMS
VOL.14,NO.3,SEPTEMBER 2009,180-192

Predicting Object-Oriented Software Quality: A Case Study

Lourdes PELAYO, Aina SADIA and Scott DICK

Abstract- **Software quality lags far behind the quality of any other engineered product that humans create, and yet software systems are essential to the modern world. Software engineers have long proposed reusing existing software components to improve overall system quality. As part of a reuse process, the reliability of a component must be established; most commonly, this is done by using a software defect-prediction model. However, research in software defect prediction suffers from a dearth of high-quality datasets that link defect rates to modern (object-oriented) software metrics. While some datasets are large (thousands of records), some contain object-oriented metrics, and some link metrics to defect counts, there are no large datasets that link object-oriented metrics to defect counts. We report on the extraction and mining of a software defect-prediction dataset based on the defect-tracking system of a very large open-source software system, the Mozilla web-browser project. We have computed 18 object-oriented software metrics for each of the over eight thousand C++ classes in the Mozilla source code, and joined these to the number of defects reported for each class. We present a statistical characterization of this rich new dataset, and report on defect prediction experiments. To overcome skewness in the dataset, we explore the use of stratification techniques.**

***Index Terms*— Software Reliability, Open Source Software, Data Mining, Knowledge Discovery in Databases, Mining Software Repositories, Machine Learning**

1. INTRODUCTION

Software systems pervade our lives. They run our cars, fly our planes, manage our power grid, and the list goes on. Our modern technological society is absolutely dependent on software systems, and software failures can easily cause economic damage, personal injury or even death. Thus, it is unsettling to realize that the quality of software systems in general lags far behind the quality of any other engineered product that humans create. Software failures currently cost the U.S. economy alone more than $78 billion per year [1]. A great many researchers have proposed re-using existing software components (which presumably are of superior quality than new components) as a way to improve programmer productivity and the overall quality of a software system [2]. However, this improvement in quality depends on the reliability of the components being reused, which is usually unknown *a priori*; hence some means of estimating the reliability of a component is needed. This is a very active area of research, as accurately estimating software reliability would greatly benefit the software industry. In this article, we focus on non-parametric models of software reliability, and in particular on machine-learning techniques. Any machine-learning approach is dependent on the quality of the data used to train the algorithm, and this is a serious shortcoming of current software reliability modeling; publicly-available datasets do not reflect current industry approaches to software design.

There are numerous approaches for acquiring a reusable software component. One could purchase commercial off-the-shelf (COTS) software [3]. Alternatively, a component could be developed in-house; many organizations have made huge investments in building up reusable component libraries, in the hopes of improving their software products and processes [4]. The emergence of the open-source software movements has also provided a new source of reusable software components. Open source involves two important properties: visible source code and a right to develop derivative works, meaning that the code can (often) be freely reused [5]. These alternatives all hold the promise of making software development quicker and cheaper, or delivering more functionality within a fixed budget and schedule. However, the quality of these components is unknown; by definition, a reused component was *not* primarily designed to be part of the new system [3, 5].

In all cases, incorporating reusable components is no mere matter of plug-and-play. A reuse process must be established, which will consider questions such as the appropriateness of the component for the new problem domain, the quality (reliability, performance, and maintainability) of the component, availability of supporting software artifacts (requirements and design documents, test plans or test suites), and the degree of reengineering required. Processes for producing software in an organization must also change when the organization commits to building up a reusable component library [4]. In this article, we focus on one particular element of the reuse process, namely determining the reliability of a module in a software component.

Software reliability is defined as the probability that a given software system or component, operating in a given environment, will not fail for a given period of time [6]. It is notoriously difficult to estimate this probability; the history of attempts stretches from the Jelinsky-Moranda model proposed in 1971 [7] through to the present day. A proxy for the actual (unknown) reliability is the predicted number of defects in a module, which is estimated using software metrics. Modules with a higher number of predicted failures require more quality improvement resources. Again, estimating this number has proven stubbornly difficult; there is no accepted form for the relationship between software metrics and reliability. An alternative approach (usually termed fault-proneness) simply attempts to model whether a module will experience a failure or not (i.e. defect prediction is converted into a binary classification problem) [8, 9]. On a

This research was supported in part by the Alberta Software Engineering Research Consortium under grant no. G230000109, in part by the Natural Sciences and Engineering Research Council of Canada under grant no. G121210906, and in part by CONACYT Mexico.

L. Pelayo, A Sadia and S. Dick are with Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB T6G 2V4 Canada (e-mails: pelayo@ece.ualberta.ca, dick@ece.ualberta.ca).

practical level, most organizations rely on a very simple Pareto analysis: they select a few software product metrics, and select the 20% of modules with the highest metric values for additional quality improvement effort. This approach does help, but numerous defects still slip through into field-deployed systems.

Evidence has shown that human experts are very poor at identifying fault-prone software modules [10]. The current state of the art in software defect prediction (defined as a regression or function-approximation problem) is to attempt to fit a statistical model (e.g. [11]) to relate a set of software metrics to the number of defects that will arise in that module. However, software defect prediction is an inexact science. Each software project has a unique combination of intellectual difficulty, size, development team composition, and third-party components, among other factors. The result is that there is no "critical value" for any software metric that definitively indicates when a module will be of lower quality. Indeed, there is not even an accepted parametric model form (e.g. linear, quadratic and Poisson regression) for predicting the number of faults in a module. Part of the problem is that there is no theory of software metrics to guide the selection of such a model form, meaning that parametric regression is reduced to a trial-and-error search through an infinite space of possible model forms. It is thus not surprising that nonparametric techniques, especially machine learning and computational intelligence, have been employed for software defect prediction. A variety of researchers have investigated neural networks [12-14], support vector machines and their hybrids [15-17], ant colony optimization [18] genetic programming [19], genetic-fuzzy systems [20], fuzzy clustering [21],[22], k-means clustering [23], classification trees [24], and classifier ensembles [25] using various defect-prediction datasets.

The defect-prediction datasets themselves have been a key stumbling block in the aforementioned research. Currently, the majority of publicly-available datasets are quite small (i.e. relatively few modules in the program), and collect only a small set of procedure-oriented metrics, meaning they do not reflect modern software development processes. In addition, some datasets do not relate the metrics for a module to any defect measure for that module; nothing beyond shotgun correlations may be found in such a dataset. Some defect-prediction datasets are based on reasonably large projects (including those archived at the PROMISE Software Engineering Repository [26], the NASA Metrics Data Program [27], and another NASA site [28]). Object-oriented (OO) metrics are collected for several datasets in [26] and [27], and several large ones in [28], while defect measures are collected for the datasets in [26] and [27]. However, at the present time we are not aware of a single publicly-available dataset that simultaneously collects object-oriented metrics and defect measures over a large number of modules (the OO metrics datasets in [26] are quite small; the KC1 dataset covers 170 classes, while the GSM class-level datasets are of a similar size). This imposes a serious limitation on defect-prediction models, as the modern practice of object-oriented development has substantially changed programming styles. Classes (the principal unit of OO functionality) tend to be made up of very few methods, and these methods in turn tend to be quite small (one study found that more than half of the methods in OO programs might contain less than five lines of executable code!) Furthermore, there tend to be a large number of interacting classes in a software system [29, 30]. The design considerations of inheritance, polymorphism and operator overloading also cause OO programs to be fundamentally different in structure than procedure-oriented programs. There is thus a crucial need for defect-prediction datasets that combine OO metrics and defect measures for large-scale modern software systems; without such data, researchers will be unable to pursue industrially relevant research.

Our primary contribution in the current paper is the extraction and mining of a new, large-scale software defect-prediction dataset, derived from a major open-source software system. Using the source code and defect-tracking database of the Mozilla project, we have created a dataset from the 8,349 C++ classes that provide the back-end functionality for the Mozilla web browser. This dataset covers over 900,000 lines of C++ code. We have computed 18 OO metrics for each class, and joined them to a defect measure extracted from the defect-tracking database (Bugzilla). The result is a dataset of 18 numeric predictor attributes, one numeric target attribute, with 8,349 records and no missing values, drawn from a modern, large-scale, industrial-quality software system. We present a statistical analysis of this dataset (statistical moments of each attribute, correlation analysis, principal components analysis). We then study defect prediction in this dataset. We were unable to effectively predict the number of defects per module, so we modified our experiments to predict fault-proneness (this has the effect of reducing variance in the class variable). To deal with the remaining skewness in the data, we employ stratification-based resampling [31]. We thus obtain a classifier that exhibits greater sensitivity to the minority (fault-prone) class, even though this is only 5% of the original dataset.

The remainder of this paper is organized as follows. We provide background on software metrics research and open-source software in Section 2. In Section 3, we discuss the Mozilla system, and the extraction of our dataset. In Section 4, we provide an extensive statistical characterization of the dataset, and perform feature reduction using principal components analysis. We describe our defect prediction experiments in Section 5, and our fault-proneness experiments in Section 6. We offer a summary and discussion of future work in Section 7.

2. BACKGROUND AND RELATED WORK

2.1 Software Metrics and Software Quality

The software metrics of interest in the current paper are formally referred to as product metrics, as they concern the actual source code. Examples are McCabe's complexity [32] or Halstead's "software science" metrics [33]. Project metrics such as the number of tests written and passed, number of open and closed problem reports among others, are beyond the scope of the current paper, as are size estimation metrics such as function or object points [34]. Product metrics express some aspect of a module (size, complexity, cohesion, coupling with other modules) as a numeric value. In the software metrics literature, each new metric is argued on philosophical and measure-theoretic grounds, and then validated on actual project data. This validation usually takes the form of a correlation or regression analysis between metric values and a defect attribute (number of defects or occurrence of defects in a module); see for instance [35, 29, 36]. Literally hundreds of product metrics have been created in the last 30 years, and there has been significant debate on the merits of these measures. One key criticism is that the measure-theoretic underpinnings of these product metrics are often shaky; it is not always clear whether the numeric values of a metric properly belong to a ratio, interval, or even an ordinal measurement scale! Furthermore, it is remarkably difficult to even attempt to determine these basic facts; notions such as "complexity," for example, are based on subjective perceptions that are not amenable to formal mathematical analysis [37].

Software product metrics also tend to be strongly correlated with each other as well as with the number of defects, a phenomenon known as multicollinearity. This is a well-known problem in the software metrics literature [9, 36], which undermines the regression models commonly used in defect prediction; these models assume independence between the predictor attributes. Principal Components Analysis (PCA) is commonly employed to overcome this problem; in addition to reducing the dimensionality of a dataset, the resulting principal components are statistically independent from one another, eliminating multicollinearity [8, 38]. Heteroskedasticity (non-uniform variance) is also a well-known problem in the software metrics literature. Furthermore, metric values on any project tend to be sharply skewed towards small values; the reason is that most modules in a software project are small and simple. The large, complex modules are rarer; however, these are where most defects tend to be found.

Beginning in 1994, researchers have turned their attention to developing metrics specifically for OO programming, in order to reflect the great differences between procedure-oriented and object-oriented code. The first and best-known OO metrics are a suite of six developed in [39] and known as the CK metrics: Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Response For a Class (RFC), Coupling Between Object Classes (CBO), and Lack of Cohesion of Methods (LCOM). Also in 1994, Lorenz and Kidd [40] proposed additional OO metrics that concentrate on inheritance and message passing in a class. Numerous other metrics have also been proposed; we refer the reader to [35, 41-44] for extensive reviews and comparisons. Recent research [45] also concentrates on collecting run-time metrics, owing to the greater usage of dynamic libraries in OO systems.

2.2 Free and Open Source Software

Open-source software is today a popular alternative to commercial software systems. There are a number of open-source systems that are widely used. The Apache Web server, the Linux operating system, and the Mozilla Firefox Web browser are all examples of open-source systems that have gained a solid foothold in their market niche; Apache is in fact the leading Web server in the global market [46]. Open-source software development is, however, extremely different from the traditional development models used in industry.

Open-source software (OSS) usually begins with a single individual or organization creating a prototype software system, and then releasing the source code on the Internet for anyone to freely use, modify and redistribute. From that point on, the original developers will usually act as a central organization, maintaining and directing the "official" version of the software; there may also be a number of "unofficial" versions maintained and developed by others [47]. The OSS development process is conducted by geographically distributed volunteer developers. There is no system-level design or detailed design and no project plan, schedule, or list of deliverables [48]. Closely related is the free software movement [49]. Owing to the great similarity in how free and open source systems are developed, researchers studying these systems will often refer simply to free or open source software (F/OSS) [50].

With very few exceptions, F/OSS developers are purely volunteers. The establishment and maintenance of a vibrant, active developer community is key to the success of a F/OSS project [48]. Members of the community are motivated to join and remain within the community for reasons of personal satisfaction, both intrinsic and extrinsic. This has important implications for the F/OSS development process. On the positive side, there is a very strong incentive to produce source code of excellent quality, both to gain recognition within the community, and for the personal satisfaction of doing well on a creative project that the developer is passionate about [51]. However, developers are much less interested in the more pedestrian aspects of software development, such as documentation and developing detailed test plans. With very few exceptions, open-source projects suffer from a lack of documentation, testing strategies that are not explicitly documented, and limited or no visibility of the system architecture. With no software contract to fulfill, there is little deadline pressure and few measurable goals, and so project schedules often slip severely. Furthermore, in spite of the oft-quoted cliché that "Given enough eyeballs, all bugs are shallow," the quality of F/OSS

projects varies widely, and no standardized quality assessment process or metrics currently exist [52]. In short, caveat emptor for anyone considering incorporating an F/OSS component into a software system. As we discuss in the next section, these disadvantages also played an important role in the development of our software quality dataset form the Mozilla project.

There is a growing body of work that studies the phenomenon of F/OSS development. These studies are motivated by the growing importance of F/OSS components in the infrastructure of the Internet and in personal and corporate IT, and are enabled by open access to the software repositories, defect-tracking systems, and mailing list archives maintained by these large projects. Probably the most influential of these is a study of Apache and Mozilla in [48], examining the mailing lists and CVS repository of both projects. Several hypotheses concerning the organization of the communities and technical aspects of the projects that lead to project success were developed. A replication of this study for FreeBSD (an operating system) is reported in [53]. The defect-tracking systems for Eclipse (an integrated development environment) and Firefox are analyzed in [54], examining the usage of the defect-tracking system, and the problems of using a world-writeable defect tracking system. These data sources are also used in [55], which compares the defect-tracking database to the CVS repository, to identify "features" (i.e. related changesets of code) within a project.

Another research direction focuses on comparing the quality of F/OSS vs. proprietary software. In [56], the source code of four major operating systems (FreeBSD, GNU/Linux, Solaris and Windows) are compared on the dimensions of file organization, code structure, code style, data organization and preprocessor utilization. No one system was dominant; for instance, Linux's code structure was seen as "excellent," but the code style lagged behind. In general, no large differences were identified. [57] also found that the maintainability of F/OSS seemed roughly equal to that of competing proprietary software. [58] examines some common beliefs about the differences between F/OSS and proprietary software: that F/OSS grows more quickly (refuted); that F/OSS projects could be more "creative" (supported); that F/OSS projects succeed because they are "simpler" (refuted); and that defects are found and fixed more rapidly in F/OSS projects (supported). The result on "simplicity" deserves extra attention; it was found that there is generally more coupling in an F/OSS project than in proprietary software. Coupling is a very serious problem as it implies that changes in one module affect logically separate modules; Briand and others [35] argue the stronger the coupling in a system, the greater the fault-proneness of the system. [59] also found that coupling in F/OSS software is a very significant problem. The claim that F/OSS software does not grow faster than proprietary software is also supported by research in [60], which refutes earlier claims in [61].

3. THE MOZILLA PROJECT

3.1 History and Development of Mozilla

The Mozilla project [62] is an open-source community dedicated to the development of a modern web browser and supporting utilities (email client, calendar, address book, internet chat and a web page composer). At the time of this writing, Mozilla produces the Firefox browser, which holds a significant share of the browser market, at about 19.2% world-wide in August 2009 [63]. However, the story of the Mozilla project reaches back seven years before the launch of Firefox, to the release of an initial version by Netscape Inc. in 1998. The early development of the community was actually quite uneven, despite the sponsorship of Netscape and its assignment of Netscape employees as full-time Mozilla developers. Mozilla was (and is) a very large software system, with a number of components that are individually larger than the entire Apache system [48]. The original architecture was deemed cumbersome, and there was a licensing requirement for proprietary libraries. Finally, it took roughly two and a half years from project inception until the first production-quality browser was launched. However, by the end of 2000, this situation was improving, with expanded documentation on the Mozilla architecture and technology (how to build and test the product), tutorials, refined processes and development tools. In fact, some of the Mozilla development tools (notably the Bugzilla defect-tracking system) have been spun off into successful open-source projects of their own [48, 62]. Today, the ongoing development of the Mozilla project is managed by the Mozilla.org staff and supported by the Mozilla Foundation.

In the Mozilla architecture, back-end functionality is provided by C++ libraries, while user interfaces are written in a mix of C++, JavaScript and XUL. The base library is XPCOM, which is written in C++, and provides the windowing engine, network communications and web page parsing. Cross-platform portability for XPCOM and other C++ libraries is provided by the Netscape Portable Runtime Environment (a virtual machine). The XPConnect libraries are built on top of XPCOM, and provides object transparency (via wrappers) between XPCOM and the Javascript user-interface code. XUL is an XML schema used to specify user-interface components. It is integrated with XPCOM and XPConnect [62].

The Bugzilla defect-tracking system is Mozilla's principal quality assurance tool. Accessible via a Web interface, it allows users anywhere in the world to report bugs in the Mozilla system. Bugzilla uses a MySQL database as its back-end. The database has individual fields recording basic data concerning each bug (serverity, operating system and bug status), while test scripts and possible patches (bug fixes) can be uploaded as "attachments." Approval of any bug fix normally requires two levels of review: by the module owner, and by a "super-reviewer" who is an expert on the affected subsystem. The change can then be committed to the CVS

source code repository, and the bug marked as fixed. However, the Bugzilla system and the Mozilla CVS repository are not integrated, and so there is no enforcement mechanism requiring that a patch be placed in Bugzilla when a change is made to the source-code repository. It is up to the individual developing the patch to actually upload it to Bugzilla. Further inquiry with the Mozilla.org staff also revealed that in cases where a solution was deemed "trivial" or involved a module not under reviewer control, patches might well be ignored, or a developer might have erroneously selected the wrong bug resolution in Bugzilla (e.g. "fixed" instead of "duplicate").

3.2 Dataset Creation

Our dataset is constructed to relate software metrics to defects found in the Mozilla project source code. We focus on extracting object-oriented software metrics from the C++ classes contained in Mozilla, and relating them to defects found in those classes and reported via the Bugzilla defect-tracking system. Creation of this dataset involves querying the Bugzilla database, extracting the defect counts, extracting software metrics from the Mozilla source code, and merging the defect counts and metric values. Our approach is similar to [64, 65], where custom scripts are used to export data from software repositories for further analysis. Our work is based on a copy of Mozilla's Bugzilla instance, donated in mid-2003. Using the Mozilla CVS source code archive, we check out the corresponding version of Mozilla for our analysis. We are interested in those bugs whose resolution is "fixed," and which have an associated patch for a C++ file. Of 190,722 reported bugs for Mozilla, 52,252 bugs are classified as "fixed," but only 4,665 of these have associated patches; fewer than 12,000 total bug reports contain patches, and not all of these are considered "fixed," as also observed in [55]. Our results only cover the "fixed" bugs with associated patches in C++ files.

The defects in Mozilla are quantified via the mechanism of deltas, which are atomic changes to a single file [48]. A single bug fix (known as a patch, and stored in a text file) could contain multiple deltas. The key pieces of information we extract from these files are the names of the source files in the CVS repository (these begin with "RCS file:") that are modified by the patch. These are extracted using Perl scripts. Each time the name of a file appears in a patch, we count this as one delta. Once all filenames have been extracted from all patches, we then count the number of occurrences of the filename, and this is the delta value for that file. We then map these file-level deltas to deltas per class. Standard coding practices usually mandate that each class definition and each method definition for that class be placed in separate files; however, we find that this is not always the case in Mozilla. In a number of cases, multiple classes are mixed into a single file, which complicates the assignment of file-level deltas to a class. Our approach is to divide the total deltas per file by the fraction of executable lines of code belonging to the different classes in the file (extracted by our metrics tool). We then sum these fractional deltas across all files that make up a single class, producing our class-level delta attribute. While this is an approximation, it is justified by prior software metrics research, which consistently finds a strong linear correlation between the number of executable lines of code and the number of defects in a code module [9].

We use the Krakatau Professional software metrics tool [66] to extract function-, file-, and class-level metrics for the code under consideration. This consists of over 935,000 lines of C++ code, in 8,349 separate classes. Our final dataset consists of the CK metrics (excluding CBO, which was uniformly 0) as well as the following metrics for each class:

- CSA: # of attributes of a class.
- CSAO: # of attributes and operations for a class.
- CSI: (NOOC * DIT) / Total # of methods.
- CSO: # of operations for a class.
- NAAC: # of attributes added relative to the superclass. For a root class NAAC= CSA.
- NAIC: # of attributes inherited from the superclass. For a root class NAIC=0.
- NOAC: # of operations added relative to the superclass. For a root class NAOC=CSO.
- NOIC: # of operations inherited from the superclass. For a root class , NOIC=0.
- NOOC: # of overridden operations in a class. For a root class, NOOC=CSO.
- NpavgC: Total # of parameters / # of methods.
- OSavg: WMC / # number of methods
- PA: # of references to private members within all methods of a class
- PPPC: Percentage of members declared public or private.

The final dataset thus contains 8,349 records, consisting of 18 OO metrics and a numeric dependent variable (Delta). We are not aware of any other software quality dataset in which a large number of OO metrics has been collected over so many classes, and associated with a defect rate of any sort. The interested reader may access this dataset at [67].

4. STATISTICAL CHARACTERIZATION

In this section, we present a statistical characterization of the Mozilla-Delta dataset. We first discuss statistical moments of the individual attributes, including skewness and kurtosis. We then examine the correlational structure of the dataset. Finally, we present a principal components analysis of the dataset, and perform feature reduction.

4.1 Statistical Moments

Table 1: Statistical Moments of the Dataset

Variable	Min	Max	Mean	Med	Std.Dev	Kurtosis	Skewness
CSA	0	108	2.580	0	6.500	6.050	55.067
CSAO	0	328	9.652	3	17.796	5.889	57.287
CSI	0	44	0.182	0	0.796	26.145	1411.309
CSO	0	274	7.068	2	13.151	6.106	59.884
DIT	0	13	0.504	0	1.117	3.314	1191.834
LCOM	0	4014	6.302	0	86.527	30.827	1191.834
NAAC	0	108	2.589	0	6.555	6.090	55.492
NAIC	0	104	1.509	0	6.774	6.726	52.201
NOAC	0	226	5.917	2	10.982	6.844	76.558
NOCC	0	186	0.327	0	2.685	45.149	2865.077
NOIC	0	430	11.994	0	48.905	5.447	31.173
NOOC	0	89	1.161	0	5.129	8.443	91.378
NPavgC	0	10	0.762	0.5	0.961	1.859	6.426
OSavg	0	33	1.530	1	1.522	5.296	44.644
PA	0	279	2.390	0	10.997	9.515	131.698
PPPC	0	150	87.740	100	26.644	-2.193	3.748
RFC	0	675	10.247	2	26.842	9.735	153.224
WMC	0	1472	15.746	3	48.945	11.432	214.463
Delta	0	104	0.140	0	1.49	44.139	2856.022

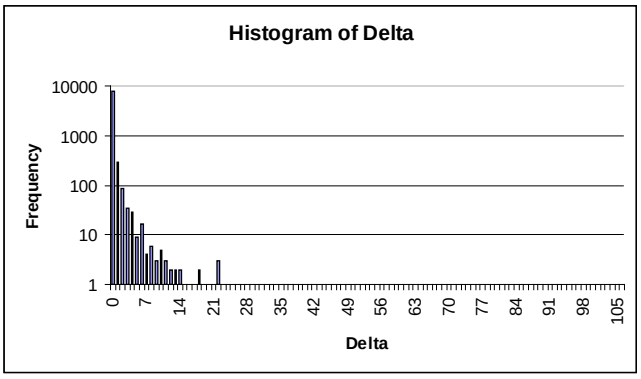

Figure 1: Semi-Logarithmic Histogram of the Delta Attribute

We present the minimum, maximum, mean, median, standard deviation, skewness and kurtosis of each individual attribute in Table 1. As can be seen, the attributes are all seriously skewed, many severely so. The skewness value should ideally (for a symmetric normal distribution) be 0; as can be seen, most attributes (with the exception of PPPC and NPavgC) have extremely high skewness values. Kurtosis is a measure of how broadly or narrowly distributed a population is about the mean. Ideally (again, a normal distribution with zero mean and unit variance) the kurtosis value should be 0; in our dataset (again with the exception of PPPC) the kurtosis values are positive, many strongly so. This indicates that the distributions are leptokurtic, or very narrow and sharply peaked. The negative kurtosis value for PPPC indicates that this attribute is platykurtic, or very broad with a flattened peak. It is particularly significant to note the high skewness and kurtosis of the Delta attribute, which is our continuous target attribute. A histogram plot of the Delta attribute (see Figure 1) also indicates a severe skewness in the data; we present the histogram on a semilogarithmic plot for convenience. These characteristics will tend to have a negative impact on data mining algorithms, and function approximation algorithms in particular [21].

4.2 Correlation Analysis

We present a correlation analysis of the predictor attributes vs. the class attribute in Table 2, using Pearson's product-moment correlation coefficient. As the reader will note, the correlations are surprisingly weak; the only exception is the CSI attribute, which is moderately correlated to Delta. In software defect prediction, one normally expects to find fairly high correlations between each attribute and the defect attribute; that is, after all, what the original empirical validations of each metric were designed to demonstrate.

In Table 3, we present pairwise correlations between all of the predictor attributes (again using Pearson's coefficient), in order to determine if multicollinearity is present in the dataset. As can be seen, the picture is mixed; some attributes correlate strongly with each other, while others correlate very weakly. We only report r-values when the corresponding p-values are less than 0.01 (i.e. r-values are significant at α=0.01); when this is not the case, we merely report this fact. From Table 3, we conclude that multicollinearity is indeed present in this dataset, but it is not universal.

Table 2: Correlation of Software Metrics to Delta

	DELTA
CSA	0.0719
CSAO	0.113
CSI	0.5044
CSO	0.0977
DIT	0.1079
LCOM	0.0479
NAAC	0.1756
NAIC	0.0486
NOAC	0.073
NOCC	0.0498
NOIC	0.0575
NOOC	0.0985
NPavgC	0.0442
OSavg	0.0363
PA	0.0685
PPPC	-0.0442
RFC	0.118
WMC	0.2001

4.3 Principal Components Analysis

We have undertaken a principal components analysis of the Mozilla-Delta dataset, using the eigenvalues of the covariance matrix. The eigenvalues are plotted in Figure 2. We select the n eigenvectors corresponding to the n largest eigenvalues as the principal components of the dataset using the Cattell scree test [68]: we look for a sharp steepening of the curve in Figure 2, and treat this as a

cutoff point. We then employ the Karhunen-Loeve transformation for feature reduction. Examination of Figure 2 indicates that eigenvectors 14 through 18 should be selected as the orthonormal basis for reducing the dataset; these five eigenvectors are presented in Table 4.

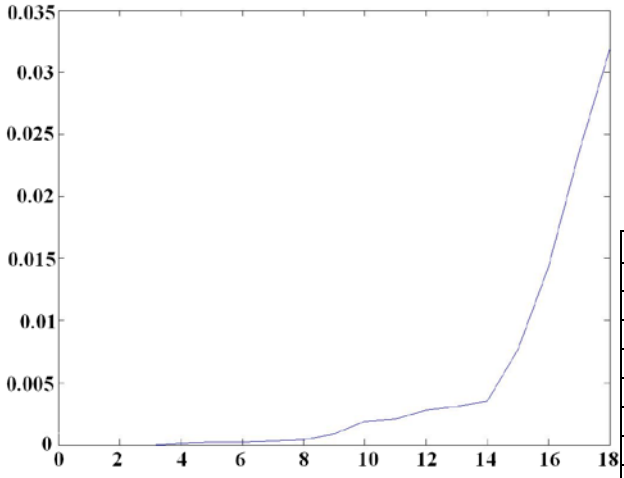

Figure 2: Eigenvalues of the Covariance Matrix

The ordering of eigenvectors in Table 4 is the same as the eigenvalues in Figure 2; the eigenvector covering the largest variance is PC5, with the amount of variance covered dropping monotonically from right to left. From Table 4, we find some interesting behaviors in the dataset when we examine the factor loadings [69]; we follow the advice in [70] and interpret loadings greater than 0.4 as being "strong." Firstly, PC5 is heavily loaded on the PPPC attribute; recall that this attribute also exhibited unique behaviors in its statistical moments. PC4 is most strongly loaded on NOIC, with cross-loading on DIT; alignment with the other attributes is weak. PC3 extensively cross-loaded; the highest loadings are on NOIC, CSA, NAAC, and CSAO, but PC3 is not strongly loaded on any single metric. PC2 is strongly loaded on NPavgC, and PC1 is strongly loaded on OSavg. What this tells us is that, for the Mozilla-Delta dataset, PPPC, NPavgC and OSavg are very nearly orthogonal to one another, while NOIC and DIT are also somewhat independent of these three.

5. DEFECT PREDICTION EXPERIMENTS

In this section, we present the results of defect prediction experiments we have performed on both the original dataset and the dataset after reduction by the Karhunen-Loeve transform. The machine-learning task in defect prediction is to estimate the number of faults likely remaining in a code module. When this technique is employed in software test management, modules with a large number of predicted defects are selected for extra quality assurance effort (i.e. more testing and debugging resources will be allocated).

We employ statistical regression (linear and PACE regression), as well as machine learning algorithms (multilayer perceptrons, radial basis function networks, and SMO regression models) in modeling the original and reduced datasets. In general, we find that this dataset is difficult for prediction algorithms; our models all showed a very high root-mean-square (RMS) testing error in our experiments. All experiments are run in the WEKA data-mining environment, using a tenfold cross-validation design. All predictor attributes were normalized to [0,1], and there are no missing attribute values; the Delta attribute was not normalized.

Table 4. Eigenvectors of the Covariance Matrix

	PC1	PC2	PC3	PC4	PC5
CSA	0.055489	-0.21135	0.349292	0.156546	0.070595
CSAO	-0.11728	-0.17425	0.329445	0.204015	0.040736
CSI	0.012005	0.001788	-0.0395	0.064328	0.00165
CSO	-0.16305	-0.1248	0.256807	0.18216	0.020671
DIT	0.11442	0.011404	-0.29214	0.445323	0.00596
LCOM	-0.02693	-0.06299	0.06726	0.032972	0.005841
NAAC	0.060699	-0.21355	0.348342	0.158285	0.073205
NAIC	-0.11	-0.06412	-0.17194	0.226948	-0.01703
NOAC	-0.12657	-0.1038	0.277501	0.13479	0.023233
NOCC	-0.00984	0.002878	0.000877	0.00434	0.000332
NOIC	0.026868	-0.16452	-0.44008	0.622703	-0.03453
NOOC	-0.18393	-0.11899	0.085896	0.219573	0.005799
NPavgC	-0.29091	0.856086	0.228383	0.31748	0.045387
OSavg	0.878041	0.228056	0.185782	0.147176	0.029306
PA	0.081914	-0.09466	0.15918	0.094145	-0.00702
PPPC	0.011142	0.007091	0.114503	0.039807	-0.99085
RFC	-0.04383	-0.09029	0.185706	0.148259	0.017187
WMC	0.108768	-0.03887	0.153319	0.107474	0.019749

The mean and standard deviation of the RMS errors for the best parameter selection we found for each model are presented in Table 5 for the original dataset, and in Table 6 for the reduced dataset. As can be seen, the average RMS errors in the tenfold cross-validation experiments are an order of magnitude greater than the average value of the Delta attribute itself (see Table 1). No method in Tables 5 or 6 is statistically superior to any other, so we only analyze one of these results in detail. Ad-hoc analysis shows that SMO regression (which is based on support vector machines) produced a small practical improvement in the prediction results in the original dataset only; notice that the resulting RMS error was also less than that of any algorithm (including SMO regression) for the reduced dataset, while the standard deviation for SMO regression is less than half that for any other method in the original dataset.

Table 3: Pairwise Correlations Between Software Metrics

CSA	CSAO	CSI	CSO	DIT	LCOM	NAAC	NAIC	NOAC	NOCC	NOIC	NOOC	NPavgC	OSavg	PA	PPPC	RFC	WMC	
1	0.8035	0.0185	0.5929	0.0593	0.3952	0.9897	*p>0.01*	0.579	*p>0.01*	0.0536	0.2798	0.1989	0.2687	0.5127	-0.135	0.4938	0.454	CSA
	1	0.1151	0.9555	0.1564	0.4717	0.7964	0.1019	0.8957	0.0327	0.1755	0.5314	0.3035	0.2817	0.5414	-0.0591	0.8246	0.7098	CSAO
		1	0.13	0.6792	*p > 0.01*	0.1079	0.3198	*p>0.01*	0.0489	0.5633	0.354	0.211	0.112	0.0737	*p>0.01*	0.1566	0.1871	CSI
			1	0.1792	0.4429	0.5849	0.1234	0.9259	0.0442	0.2111	0.581	0.3127	0.2483	0.479	*p>0.01*	0.8713	0.7325	CSO
				1	*p > 0.01*	0.0754	*0.452*	0.0524	0.0291	0.7791	0.3528	0.2743	0.151	0.092	*p>0.01*	0.2085	0.1684	DIT
					1	0.3916	*p>0.01*	0.388	*p>0.01*	0.0488	0.3048	0.0406	0.0778	0.3542	*p>0.01*	0.4811	0.3961	LCOM
						1	0.0309	0.5721	*p>0.01*	0.0505	0.2764	0.1959	0.2659	0.5083	-0.1412	0.4926	0.4681	NAAC
							1	0.0367	*p>0.01*	0.5367	0.2374	0.1269	0.0418	0.0493	0.0392	0.1101	0.0783	NAIC
								1	0.0323	0.0719	0.2339	0.2743	0.2327	0.3767	*p>0.01*	0.7939	0.6778	NOAC
									1	0.0323	0.0447	0.0436	*p>0.01*	*p>0.01*	*p>0.01*	0.046	0.0401	NOCC
										1	0.3868	0.1974	0.0991	0.1061	0.0466	0.2377	0.1605	NOIC
											1	0.2184	0.138	0.4213	*p>0.01*	0.5374	0.4272	NOOC
												1	0.3605	0.1434	-0.0401	0.2801	0.2752	NPavgC
													1	0.2868	-0.0381	0.3159	0.5134	OSavg
														1	0.0771	0.5315	0.541	PA
															1	*p>0.01*	-0.053	PPPC
																1	0.861	RFC
																	1	WMC

Table 5. Regression on Full Dataset

Model	Mean RMS	Std. Dev. RMS
Linear Regression	1.3487	0.8495
PACE Regression	1.4608	1.6755
Radial Basis Function Network	1.1677	0.9412
Multilayer Perceptron	1.2242	0.9201
SMO Regression	1.0133	0.4079

Table 6. Regression on Reduced Dataset

Model	Mean RMS	Std. Dev. RMS
Linear Regression	1.1954	0.9213
PACE Regression	1.1954	0.9212
Radial Basis Function Network	1.1966	0.9067
Multilayer Perceptron	1.2000	0.9203
SMO Regression	1.1541	0.9966

Table 7. Actual vs. Predicted Deltas R^2

Partition	R^2
1	0.0007
2	0.0026
3	0.0144
4	0.0009
5	0.8904
6	0.00003
7	0.0547
8	0.0418
9	0.0118
10	0.0043

(a)

Partition	R^2
1	0.0008
2	0.0221
3	0.0002
4	0.0300
5	0.0504
6	0.0535
7	0.0005
8	0.0033
9	0.0036
10	0.0334

(b)

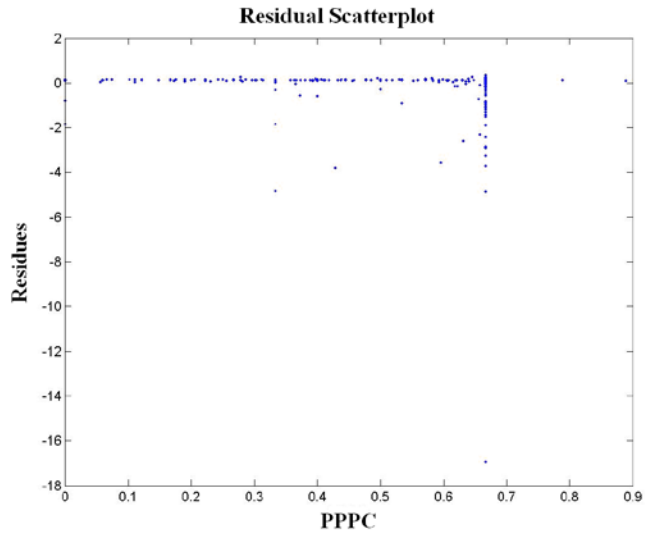

Figure 3: PPPC vs. Residuals, Full Dataset, Partition 10, SMO Regression

We now investigate the SMO prediction errors (residuals) in detail. Tables 7(a) and (b) provide the r^2 values for each partition of the tenfold cross-validation experiments on the original and reduced datasets, respectively. These are computed as the square of the Pearson's r between the actual and predicted values of the Delta attribute. Plainly, the prediction errors are generally poor, with one exception in the 5th partition of the full dataset.

Recall from Section 4 that the maximum value of the Delta attribute was 104.0. In the SMO experiment on the full dataset, this record was assigned to the test set in partition 5, and was in fact predicted relatively well; the prediction for that record was 58.272 (residual = -45.728), meaning that this record was recognized as having a very high value. However, this was the exception rather than the rule. The predictions for virtually every record were close to 0, meaning that any record with a Delta value much different than 0 was seriously under-predicted. The residue for the record with Delta=104 is the only one to break this pattern. As Pearson's coefficient is a product-moment correlation, we suspect that this one reasonable prediction biased the entire correlation for this partition. All remaining partitions, in both the original and reduced datasets, show the same trend in the residues. This means that the SMO algorithm (the most accurate we found) was unable to recognize records with a Delta value differing from 0; as these are precisely the software modules a developer is interested in detecting, we must conclude that our defect-prediction experiments have been ineffective.

There was one exception to this finding, for the PPPC metric. Recall from Section 4 that the statistical moments of this attribute are different from the other attributes, and that the eigenvector covering the greatest variance in this dataset loads very heavily on this attribute. We present a typical residue plot for this attribute in Figure 3, where we see a vertical linear structure that consumes most of the under-prediction errors. Importantly, this linear structure is found at high values of the PPPC attribute. It would thus seem reasonable to suggest that we could use a ranking on the PPPC value to identify high-risk software modules in this dataset with a reasonable degree of accuracy; modules in the top k%, ranked by the PPPC attribute, should be considered high-risk and receive additional quality-assurance effort.

6. FAULT-PRONENESS EXPERIMENTS

In Section 5 we saw that treating defect prediction as a function approximation problem was generally ineffective, as every algorithm tended to simply predict a value of "0." We believe that this is due to a combination of heteroskedasticity and skewness in the data, and so in this section we attempt to reduce the effect of these problems in the original dataset. We reduce the variance of the Delta attribute by converting it to a binary class (i.e. we convert the defect-prediction problem to a fault-proneness problem). We then use stratification-based resampling to reduce the impact of skewness on this dataset, following [71-74]; in particular we use uniform undersampling and the SMOTE oversampling algorithm [31]. SMOTE produces synthetic examples of the minority class, in order to avoid the tightly-focused decision regions that occur when minority-class samples are merely replicated. This

combination of techniques should lead to better fault predictions.

The experiments are performed using several classification algorithms implemented in WEKA [75]: Support Vector Machines, KStar instance-based classifier, RIPPER, Random Forests, Radial Basis Functions and decision trees [76]. As overall accuracy is a poor measure of classifier performance on imbalanced data, we use Cohen's Kappa statistic [77] computed from the confusion matrices of each experiment. The Kappa statistic is intended to represent the chance-corrected agreement between two raters. In highly skewed datasets, where classifiers may simply predict the majority class for all instances, we believe that this statistic will be more revealing than classification accuracy. All experiments follow a tenfold cross-validation design, and we report the average and standard deviation of the statistics for the best parameterization of each algorithm.

We first present the results of fault-proneness modeling without stratification in Table 8. We include both the accuracy and Kappa statistic to demonstrate how misleading classification accuracy is in this case. Every method achieves 92-94% accuracy, but this is roughly the fraction of the dataset belonging to the majority class! The minority class is not being predicted well at all; this fact shows up clearly in the Kappa statistic, which is not greater than 0.1233 for any method. The skewness in the dataset is severely biasing the classifiers, and so we will next proceed with our stratification experiments.

Table 8. Results for original distribution

Algorithm	Accuracy	Std Dev (accuracy)	Kappa	Std Dev (Kappa)
J48 Decision Tree	94.01	0.0548	0.0066	0.0145
J48 Decision Tree (unpruned)	93.23	0.4623	0.0791	0.0468
RIPPER	93.89	0.2319	0.0041	0.0130
Kstar	92.13	0.5429	0.1233	0.0559
Radial Basis Function Nets	94.02	0.0356	0.0000	0.0000
Random Forests	93.36	0.4122	0.1022	0.0656
Support Vector Machines	94.02	0.0356	0.0000	0.0000

We use different rates of over and under-sampling to modify the distribution of the classes in the training dataset of each fold in our cross-validation. We employ 25 resampling strategies, each one being a class-by-class specification of the degree of under-sampling or over-sampling to be applied. The rates of over-sampling are 0, 100, 200, 300, 400 and 500% representing the number of synthetic minority examples (as a percentage of the original minority class size) added to the training set. For under-sampling we remove 0, 20, 40, 60 and 80% of the examples from the majority class in the training set. We report the average and standard deviation of the Kappa statistic on the (unchanged) test data set for each strategy on the best parameterization of that algorithm in Tables 9-15. The highest Kappa statistic value obtained is highlighted in bold.

Table 9. J48 Decision trees

	0	100	200	300	400	500
0	0.0066 ±0.015	0.0066 ±0.015	0.0723 ±0.022	0.0896 ±0.063	0.0897 ±0.052	0.1090 ±0.102
20	0.0277 ±0.031	0.0149 ±0.019	0.1019 ±0.047	0.1254 ±0.059	0.1119 ±0.059	**0.1307** ±0.061
40	0.0285 ±0.031	0.0270 ±0.028	0.0965 ±0.055	0.1018 ±0.053	0.1139 ±0.048	0.1216 ±0.067
60	0.0683 ±0.051	0.0375 ±0.038	0.1212 ±0.049	0.1028 ±0.053	0.1106 ±0.046	0.1204 ±0.038
80	0.1233 ±0.038	0.0996 ±0.052	0.1023 ±0.026	0.0948 ±0.027	0.0992 ±0.029	0.0966 ±0.022

Table 10. J48 Decision trees (unpruned)

	0	100	200	300	400	500
0	0.0791 ±0.047	0.0791 ±0.047	0.0805 ±0.033	0.1073 ±0.062	0.1060 ±0.053	0.1148 ±0.082
20	0.0986 ±0.053	0.0714 ±0.046	0.1016 ±0.056	0.1145 ±0.063	0.1168 ±0.057	0.1323 ±0.062
40	0.0727 ±0.033	0.0909 ±0.044	0.1020 ±0.066	0.1095 ±0.049	0.1173 ±0.055	0.1311 ±0.068
60	0.1097 ±0.044	0.0889 ±0.057	**0.1324** ±0.044	0.1032 ±0.059	0.1119 ±0.046	0.1222 ±0.032
80	0.1102 ±0.049	0.1008 ±0.048	0.1023 ±0.025	0.0944 ±0.026	0.0966 ±0.031	0.1000 ±0.022

Table 11. RIPPER

	0	100	200	300	400	500
0	0.0041 ±0.013	0.0131 ±0.024	0.0431 ±0.047	0.0270 ±0.028	0.0351 ±0.030	0.0232 ±0.019
20	0.0152 ±0.036	0.0039 ±0.016	0.0333 ±0.038	0.0469 ±0.043	0.0462 ±0.071	0.0353 ±0.027
40	0.0153 ±0.030	0.0037 ±0.015	0.0601 ±0.035	0.0820 ±0.055	0.0417 ±0.036	0.0825 ±0.063
60	0.0279 ±0.040	0.0375 ±0.047	0.1027 ±0.044	**0.1129** ±0.060	0.0930 ±0.031	0.0883 ±0.037
80	0.0743 ±0.052	0.1039 ±0.058	0.0999 ±0.028	0.0815 ±0.018	0.0819 ±0.015	0.0831 ±0.017

In Tables 9-15, we are always able to improve the classification results with some combination of under- and over-sampling. However, the "best" resampling strategy changes with each algorithm, as does the relative improvement. KStar still produces the most balanced classifications, but is still far from the theoretical limit of 1.0 (for a perfect classification). In summation the Mozilla-Delta dataset is still a difficult one to accurately model, due at least in part to the very high level of skewness.

Table 12. KStar

	0	100	200	300	400	500
0	0.1258	0.1258	0.1382	0.1352	0.1373	0.1381
	±0.049	±0.049	±0.059	±0.053	±0.063	±0.053
20	0.1267	0.1334	0.1488	0.1417	0.1439	0.1468
	±0.049	±0.052	±0.063	±0.057	±0.054	±0.057
40	0.1333	0.1332	0.1322	0.1327	0.1517	0.1433
	±0.044	±0.056	±0.038	±0.065	±0.047	±0.052
60	0.1512	**0.1648**	0.1389	0.1341	0.1217	0.1149
	±0.053	±0.057	±0.054	±0.047	±0.039	±0.044
80	0.1348	0.1247	0.1067	0.1093	0.1013	0.1017
	±0.062	±0.037	±0.033	±0.026	±0.029	±0.023

Table 13. Radial Basis Function Networks

	0	100	200	300	400	500
0	0.0000	0.0000	0.0000	0.0000	-0.0002	0.0000
	±0.000	±0.000	±0.000	±0.000	±0.001	±0.000
20	0.0000	0.0000	0.0000	0.0000	-0.0005	-0.0007
	±0.000	±0.000	±0.000	±0.000	±0.001	±0.002
40	0.0000	0.0000	0.0000	-0.0011	0.0079	0.0239
	±0.000	±0.000	±0.000	±0.003	±0.024	±0.032
60	0.0000	0.0000	0.0018	0.0370	**0.0815**	0.0577
	±0.000	±0.000	±0.014	±0.055	±0.046	±0.026
80	0.0021	0.0018	0.0512	-0.0093	-0.0016	0.0001
	±0.013	±0.022	±0.024	±0.011	±0.004	±0.000

Table 14. Random Forests

	0	100	200	300	400	500
0	0.1022	0.0856	0.1185	0.1225	0.1060	0.1217
	±0.066	±0.063	±0.090	±0.047	±0.050	±0.051
20	0.1141	0.1099	0.1126	0.1221	0.1177	0.1373
	±0.068	±0.069	±0.067	±0.076	±0.069	±0.082
40	0.1055	0.1132	0.0994	0.1150	0.1190	0.1361
	±0.056	±0.081	±0.083	±0.055	±0.058	±0.068
60	0.1202	0.1331	0.1203	0.1364	0.1402	0.0986
	±0.109	±0.075	±0.064	±0.042	±0.049	±0.074
80	**0.1467**	0.1337	0.1133	0.1353	0.1126	0.1193
	±0.047	±0.018	±0.048	±0.038	±0.031	±0.041

Table 15. Support Vector Machines

	0	100	200	300	400	500
0	0.1088	0.1088	0.1243	0.1217	0.1421	0.1269
	±0.038	±0.038	±0.051	±0.065	±0.043	±0.036
20	0.1150	0.1233	**0.1514**	0.1367	0.1200	0.1331
	±0.052	±0.043	±0.047	±0.037	±0.035	±0.048
40	0.0999	0.1188	0.1192	0.1137	0.1293	0.1454
	±0.046	±0.064	±0.041	±0.050	±0.033	±0.047
60	0.1222	0.1237	0.1356	0.1298	0.1136	0.1222
	±0.036	±0.048	±0.046	±0.027	±0.028	±0.028
80	0.1409	0.1303	0.1104	0.1067	0.1004	0.1117
	±0.063	±0.038	±0.021	±0.016	±0.014	±0.017

7. CONCLUSIONS

We have reported on the extraction and mining of a new software defect-prediction dataset, drawn from the Mozilla defect-tracking database and CVS repository. This dataset is, to the best of the authors' knowledge, unique among publicly-available datasets in its combination of size, use of object-oriented metrics, and incorporation of a defect measure. We have reported on a statistical analysis of, and predictive modeling for, this dataset. Heteroskedasticity and skewness appear to be significant problems within this dataset, as with other defect-prediction datasets. We have pursued defect-prediction and fault-proneness studies in this dataset, using statistical and machine-learning methods. We found that defect prediction led to generally poor results; however, we did find that ranking modules using the PPPC attribute would lead to improved predictions. Likewise, we found that using stratification techniques led to improved predictions, as measured by the Kappa statistic. As usual in resampling studies, this means that an improvement in the minority-class prediction was achieved at the cost of increased errors for the majority class. However, in the software domain, the price of allowing faults to slip through into field usage is so high that we believe this tradeoff is advantageous.

In future work, our analysis could also be combined with additional evidence from other software artifacts. The study by Mockus et al. [48] examined several other artifacts in the Mozilla and Apache development communities (the source code repository, mailing list archives). This possibility should be investigated on a mix of large-scale open-source projects (e.g. Mozilla, Apache, Eclipse, Gnome, the BSD operating systems and Linux) as well as smaller-scale projects. The Sourceforge.net (http://sourceforge.net/index.php) hosting site presently houses roughly 150,000 open-source projects of varying size and maturity. Developing an integrated methodology for examining these projects and their communities could be very useful to software engineers considering an open-source component. We will also pursue software reliability growth modeling using reliability growth data extracted from the Mozilla defect-tracking database.

ACKNOWLEDGEMENT

We would like to thank the Mozilla Foundation for providing a copy of their defect-tracking database for this research.

REFERENCES

[1] M. Levinson, "Let's Stop Wasting $78 Billion per Year," in *CIO Magazine* October 15, 2001.

[2] N. Y. Lee and C. R. Litecky, "An empirical study of software reuse with special attention to Ada," *IEEE Trans. Soft. Eng.*, vol. 23, pp. 537-549, 1997.

[3] S. Sedigh-Ali, A. Ghafoor, and R. Paul, "Software Engineering Metrics for COTS Based Systems," *IEEE Computer*, vol. 34, pp. 44-50, 2001.

[4] M. Morisio, M. Ezran, and C. Tully, "Success and Failure Factors in Software Reuse," *IEEE Trans. Soft. Eng.*, vol. 28, pp. 340-357, 2002.

[5] D. Spinellis and C. Szyperski, "How is open source affecting software development?," *Software, IEEE*, vol. 21, pp. 28-33, 2004.

[6] M. A. Friedman and J. M. Voas, *Software Assessment: Reliability, Safety, Testability*. New York, NY: John Wiley & Sons, Inc., 1995.

[7] Z. Jelinski and P. B. Moranda, "Software reliability research," in *Proc. Statistical Computer Performance Evaluation*, Providence, RI, USA, 1971, pp. 465-484.

[8] K. El Emam, "A methodology for validating software product metrics," National Research Council of Canada, Technical Report NRC/ERB-1076 2000.

[9] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai, "The confounding effect of class size on the validity of object-oriented metrics," *IEEE Trans. Soft. Eng.*, vol. 27, pp. 630-650, 2001.

[10] P. Tomaszewski, J. Håkansson, H. Grahn, and L. Lundberg, "Statistical models vs. expert estimation for fault prediction in modified code - an industrial case study," *Journal of Systems and Software*, vol. 80, pp. 1227-1238, 2007.

[11] T. M. Khoshgoftaar and K. Gao, "Count models for software quality estimation," *IEEE Trans. Rel.*, vol. 56, pp. 212-222, 2007.

[12] I. Gondra, "Applying machine learning to software fault-proneness prediction," *Journal of Systems and Software*, vol. 81, pp. 186-195, 2008.

[13] Q. P. Hu, Y. S. Dai, M. Xie, and S. H. Ng, "Early software reliability prediction with extended ANN model," in *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC '06)*, Sept 2006, pp. 234-239.

[14] M. M. T. Thwin and T.-S. Quah, "Application of neural networks for software quality prediction using object-oriented metrics," *Journal of Systems and Software*, vol. 76, pp. 147-156, 2005.

[15] Y. Bo and L. Xiang, "A study on software reliability prediction based on support vector machines," in *Industrial Engineering and Engineering Management, 2007 IEEE International Conference on*, 2007, pp. 1176-1180.

[16] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *Journal of Systems and Software*, vol. 81, pp. 649-660, 2008.

[17] P.-F. Pai and W.-C. Hong, "Software reliability forecasting by support vector machines with simulated annealing algorithms," *Journal of Systems and Software*, vol. 79, pp. 747-755, 2006.

[18] O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer, and R. Haesen, "Mining software repositories for comprehensible software fault prediction models," *Journal of Systems and Software*, vol. 81, pp. 823-839, 2008.

[19] T. M. Khoshgoftaar, M. P. Evett, E. B. Allen, and P.-D. Chien, "An application of genetic programming to software quality prediction," in *Computational Intelligence in Software Engineering*, J. F. P. W. Pedrycz, Ed. River Edge, NJ, USA: World Scientific, 1998, pp. 176-195.

[20] E. Baisch and T. Liedtke, "Comparison of conventional approaches and soft-computing approaches for software quality prediction," in *Proc. Int. C. Syst., Man, Cybern.*, Orlando, FL, USA, 1997, pp. 1045-1049.

[21] S. Dick, A. Meeks, M. Last, H. Bunke, and A. Kandel, "Data Mining in Software Metrics Databases," *Fuzzy Sets and Systems*, vol. 145, pp. 81-110, 2004.

[22] X. Yuan, T. M. Khoshgoftaar, E. B. Allen, and K. Ganesan, "An application of fuzzy clustering to software quality prediction," in *Proc. IEEE Symp. App. Spc. Soft. Eng. Tech.*, Richardson, TX, USA, 2000, pp. 85-90.

[23] N. Seliya and T. M. Khoshgoftaar, "Software quality analysis of unlabeled program modules with semisupervised clustering," *IEEE Trans. Syst., Man and Cybern., Part A*, vol. 37, pp. 201-211, 2007.

[24] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl, "Classification-tree models of software-quality over multiple releases," *IEEE Trans. Rel.*, vol. 49, pp. 4-11, 2000.

[25] N. Raj Kiran and V. Ravi, "Software reliability prediction by soft computing techniques," *Journal of Systems and Software*, vol. 81, pp. 576-583, 2008.

[26] G. Boetticher, T. Menzies, and T. Ostrand, "PROMISE Repository of empirical software engineering data," West Virginia University, Department of Computer Science: http://promisedata.org/repository/, 2008.

[27] C. McNemar, "NASA/WVU IV&V Facility Metrics Data Program," NASA Independent Verification & Validation Facility: http://mdp.ivv.nasa.gov, 2004.

[28] Staff, "OO Java metrics," NASA Software Technology Assurance technology Center: http://satc.gsfc.nasa.gov/metrics/codemetrics/oo/java/index.html, 1999.

[29] D. Glasberg, K. El-Emam, W. Melo, and N. Madhavji, "Validating object-oriented design metrics on a commercial Java application," National Research Council of Canada, Technical Report NRC 44146 2000.

[30] N. Wilde, P. Matthews, and R. Huitt, "Maintaining object-oriented software," *IEEE Software*, vol. 10, pp. 75-80, 1993.

[31] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Syntethic Minority Over-Sampling Technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321-357, June 2002.

[32] T. J. McCabe, "A complexity measure," *IEEE Trans. Soft. Eng.*, vol. 2, pp. 308-320, 1976.

[33] M. Halstead, *Elements of Software Science*. New York, NY, USA: Elsevier, 1977.

[34] B. Boehm, B. Clark, E. Horowitz, and C. Westland, "Cost models for future software life cycle processes: COCOMO 2.0," *Annals of Software Engineering*, vol. 1, pp. 57-94, 1995.

[35] L. C. Briand, J. Daly, V. Porter, and J. Wust, "A comprehensive empirical validation of design measures for object-oriented systems," in *Proc. Int. Soft. Metrics. Symp.*, Bethesda, MD, USA, 1998, pp. 246-257.

[36] R. Harrison, S. Counsell, and R. Nithi, "Coupling metrics for object-oriented design," in *Int. Soft. Metrics Symp.*, Bethesda, MD, USA, 1998, pp. 150-157.

[37] L. C. Briand, K. El Emam, and S. Morasca, "On the application of measurement theory in software engineering," International Software Engineering Research Network, Technical Report ISERN-95-04 1995.

[38] J. C. Munson and T. M. Khoshgoftaar, "Software metrics for reliability assessment," in *Handbook of Software Reliability Engineering*, M. R. Lyu, Ed. New York, NY: McGraw-Hill, 1996, pp. 167-254.

[39] S. R. Chidamber and F. C. Kemerer, "A metrics suite for object-oriented design," *IEEE Trans. Soft. Eng.*, vol. 20, pp. 476-493, 1994.

[40] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics*. Englewood Cliffs, NJ, USA: Prentice Hall, 1994.

[41] L. C. Briand, J. Daly, and J. Wust, "A unified framework for cohesion measurement in object-oriented systems," *Empirical Software Engineering*, vol. 3, pp. 65-117, 1998.

[42] L. C. Briand, J. Daly, and J. Wust, "A unified framework for coupling measurement in object-oriented systems," *IEEE Trans. Soft. Eng.*, vol. 25, pp. 91-121, 1999.

[43] S. Counsell, S. Swift, and J. Crampton, "The interpretation and utility of three cohesion metrics for object-oriented design," *ACM Trans. Soft. Eng. Method.*, vol. 15, pp. 123-149, 2006.

[44] T. M. Meyers and D. Binkley, "An empirical study of slice-based cohesion and coupling metrics," *ACM Transactions on Software Engineering and Methodology*, vol. 17, pp. 2:1-2:27, 2008.

[45] E. Arisholm, L. C. Briand, and A. Foyen, "Dynamic coupling measurement for object-oriented software," *IEEE Trans. Soft. Eng.*, vol. 30, pp. 491-506, 2004.

[46] Staff, "Netcraft: Web Server Survey Articles," Netcraft Ltd.: http://news.netcraft.com/archives/web_server_survey.html, 2009.

[47] E. Raymond, "The cathedral and the bazaar," in *Linux Kongress*, Wurzburg, Germany, 1997.

[48] A. Mockus, T. Roy, and J. D. Herbsleb, "Two case studies of open source software development: Apache and Mozilla," *ACM Trans. Soft. Eng. Meth.*, vol. 11, pp. 309-346, 2002.

[49] Staff, "FSF – The Free Software Foundation," Free Software Foundation, Inc.: http://www.fsf.org, 2007.

[50] G. Madey, V. Freeh, and R. Tynan, "Modeling the free/open source software community: a quantitative investigation," in *Free/Open Source Software Development*, S. Koch, Ed. Hershey, PA, USA: Idea Group Publishing, 2005.

[51] K. R. Lakhani and R. Wolf, "Why hackers do what they do: understanding motivation and effort in free/open source software

projects," in *Perspectives on Free and Open Source Software*, J. Feller, B. Fitzgerald, S. A. Hissam, and K. R. Lakhani, Eds. Cambridge, MA, USA: MIT Press, 2005, pp. 3-22.
[52] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, "Code quality analysis in open source software development," *Info. Sys. J.*, vol. 12, pp. 43-60, 2002.
[53] T. Dinh-Trong and J. M. Bieman, "Open source software development: a case study of FreeBSD," in *Proc. Int. Symp. Software Metrics*, Chicago, IL, USA, 2004, pp. 96-105.
[54] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *Proc. OOPSLA Wkshp Eclipse Technology eXchange*, San Diego, CA, USA, 2005, pp. 35-39.
[55] M. Fischer, M. Pinzger, and H. Gall, "Analyzing and relating bug report data for feature tracking," in *Proc. Working C. Reverse Eng.*, Victoria, BC, Canada, 2003, pp. 90-99.
[56] S. Diomidis, "A tale of four kernels," in *Proceedings of the 30th international conference on Software engineering* Leipzig, Germany: ACM, 2008.
[57] S. Ioannis, S. Ioannis, A. Lefteris, and O. Apostolos, "Open source software development should strive for even greater code maintainability." vol. 47: ACM, 2004, pp. 83-87.
[58] J. W. Paulson, G. Succi, and A. Eberlein, "An empirical study of open-source and closed-source software products," *Software Engineering, IEEE Transactions on*, vol. 30, pp. 246-256, 2004.
[59] L. Yu, S. R. Schach, K. Chen, G. Z. Heller, and J. Offutt, "Maintainability of the kernels of open-source operating systems: A comparison of Linux with FreeBSD, NetBSD, and OpenBSD," *Journal of Systems and Software*, vol. 79, pp. 807-815, 2006.
[60] I. Clemente and B. James, "The evolution of FreeBSD and linux," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering* Rio de Janeiro, Brazil: ACM, 2006.
[61] M. Godfrey and Q. Tu., "Evolution in Open Source Software: A Case Study," in *Proc. of the 2000 Intl. Conference on Software Maintenance (ICSM-00)*, San Jose, California, 2000.
[62] Mozilla.org, "Mozilla.org – Home of the Mozilla Project," The Mozilla Foundation: http://www.mozilla.org, 2006.
[63] Staff, "Browser market share white paper," Janco Associates Inc.: http://www.e-janco.com/browser.htm, 2009.
[64] O. Alonso, P. T. Devanbu, and M. Gertz, "Database techniques for the analysis and exploration of software repositories," in *Proc. Int. Wkshp Mining Software Repositories*, Edinburgh, Scotland, 2004, pp. 37-41.
[65] S. McLellan, A. Roesler, Z. Fei, S. Chandran, and C. Spinuzzi, "Experience using web-based shotgun measures for large-scale system characterization and improvement," *IEEE Trans. Soft. Eng.*, vol. 24, pp. 268-277, 1998.
[66] Staff, "Power Software - Products - Krakatau Professional," Power Software: http://www.powersoftware.com/kp/, 2008.
[67] S. Dick, "Mozilla-Delta Dataset," University of Alberta: http://www.ece.ualberta.ca/~dick/datasets/Mozilla.csv, 2006.
[68] R. B. Cattell, "The Scree Test for the Number of Factors," *Multivariate Behavioral Research*, vol. 1, pp. 245-276, 1966.
[69] Staff, "Principal Components and Factor Analysis," in *The Statistics Homepage* StatSoft, Inc.: http://www.statsoft.com/textbook/stathome.html?stfacan.html&1, 2008.
[70] J. E. Raubenheimer, "An item selection procedure to maximize scale reliability and validity," *South African Journal of Industrial Psychology*, vol. 30, pp. 59-64, 2004.
[71] C.-P. Chang and C.-P. Chu, "Defect prevention in software processes: An action-based approach," *Journal of Systems and Software*, vol. 80, pp. 559-570, 2007.
[72] S. Dick and A. Kandel, "The Use of Resampling in Software Metrics Datasets," in *Artificial Intelligence Methods in Software Testing*, M. Last, A. Kandel, and H. Bunke, Eds. Singapore: World Sci. Pub. Co., 2004, pp. 175-208.
[73] K. Kaminsky and G. D. Boetticher, "Knowledge-based techniques for software quality management," in *Proceeding (430) Knowledge Sharing and Collaborative Engineering*, St. Thomas, US Virgin Islands, 2004.
[74] L. Pelayo and S. Dick, "Applying Novel Resampling Strategies to Software Defect Prediction," in *Proceedings, NAFIPS 2007*, San Diego, CA, USA, 2007, pp. 69-72.
[75] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*, 2nd ed. San Francisco: Morgan Kaufmann, 2005.
[76] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*, 2 ed.: Morgan Kauffman, 2001.
[77] J. Cohen, "A Coefficient of Agreement for Nominal Scales," *Educational and Psychological Measurement*, vol. 20, pp. 37-46, 1960.

Lourdes Pelayo received B.S. degree in digital systems and communications engineering in 2000 and MBA in 2003 from University of Juarez, Mexico. She is currently pursuing her PhD in Computer Engineering at the University of Alberta, Canada.

She has participated in the technical committee of FUZZ-IEEE 2008 and in the technical review committee of IFSA/EUSFLAT09.

Aina Sadia received the M.Sc. degree in Computer Engineering from the University of Alberta in 2005. She is currently a Management Consultant for the City of Edmonton.

Scott Dick received the B.Sc. degree and the M.Sc. degree in computer science in 1997 and 1999, and his Ph.D. in computer science & engineering in 2002, all from the University of South Florida. His Ph.D. dissertation received the USF Outstanding Dissertation Prize in 2003.

From 2002 to 2008 he was an Assistant Professor of Electrical and Computer Engineering at the University of Alberta in Edmonton, AB. Since 2008, he has been an Associate Professor in the same department and has published over 40 scientific articles in journals and conferences.

Dr. Dick is a member of the IEEE Computational Intelligence Society's Fuzzy Systems Technical Committee. He is a member of the ACM, IEEE, ASEE, and an associate member of Sigma Xi.