Understanding the Impact of Code and Process Metrics on Post-release Defects: A Case Study on the Eclipse Project

Emad Shihab, Zhen Ming Jiang, Walid M. Ibrahim, Bram Adams and Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University, Kingston, ON, Canada
{emads, zmjiang, walid, bram, ahmed}@cs.queensu.ca

ABSTRACT

Research studying the quality of software applications continues to grow rapidly with researchers building regression models that combine a large number of metrics. However, these models are hard to deploy in practice due to the cost associated with collecting all the needed metrics, the complexity of the models and the black box nature of the models. For example, techniques such as PCA merge a large number of metrics into composite metrics that are no longer easy to explain. In this paper, we use a statistical approach recently proposed by Cataldo *et al.* to create explainable regression models. A case study on the Eclipse open source project shows that only 4 out of the 34 code and process metrics impacts the likelihood of finding a post-release defect. In addition, our approach is able to quantify the impact of these metrics on the likelihood of finding post-release defects. Finally, we demonstrate that our simple models achieve comparable performance over more complex PCA-based models while providing practitioners with intuitive explanations for its predictions.

1. INTRODUCTION

A large portion of software development costs is spent on maintenance and evolution activities [1, 2]. Fixing software defects is one area that takes up a large amount of this maintenance effort. Therefore, practitioners and managers are always looking for ways to reduce the bug fixing effort. In particular, they are interested in identifying which parts of the software contain defects, to achieve short term goals, such as prioritizing the testing efforts for the following releases. In addition, practitioners are interested in understanding the main factors that impact these defects, to achieve longer term goals, such as driving process improvement initiatives to deliver better quality software.

An extensive body of work has focused on finding the fault-prone locations in software systems. The majority of this work builds prediction models that predict where future defects are likely to occur. The work varies in terms of the domains covered (i.e., open source [3, 4] vs. commercial software [5–7]), in terms of the metrics used to predict the defects (i.e., using process [3] vs. code metrics [4]) and in terms of the types of defects it aims to predict for (i.e., pre-release [6] vs. post-release [4] or both [8]).

At the same time, these prediction models are becoming more and more complex over time. New studies are investigating more aspects that may help improve prediction accuracy, which leads to more metrics (i.e., independent variables) being input to the prediction models.

Although adding more metrics (i.e., independent variables) to the prediction models may increase the overall prediction accuracy, it also introduces some negative side effects. First, it makes the models more complex and therefore, makes them less desirable to adopt in practical settings. Second, adding more independent variables to the prediction model makes determining which independent variables *actually* produce the effect (i.e., impact) on post-release defects more complicated (due to multicollinearity [9]). The aforementioned problems turn the complex prediction models into black-box solutions that can be used to know where the defects are, but do not provide insight into the underlying reasons for what impacts the defects. The black-box nature of such models bug prediction is a major hurdle in the adoption of these models in practice.

The goal of our study is to use the code and process metrics previously used to build complex prediction models and to narrow this set of metrics to a much smaller number that can be used in a logistic regression model to *understand what impacts* post-release defects. Understanding what impacts post-release defects can be leveraged by practitioners to drive process changes, build better tools (e.g., monitoring tools) and drive future research efforts.

We apply a statistical approach, recently proposed by Cataldo *et al.* [5], to identify statistically significant and minimally collinear metrics (i.e., independent variables in a logistic regression model) that impact post-release defects. In addition, we use odds ratios to quantify the impact of these independent variables on the dependent variable, post-release defects. For example, we quantify the increase in the likelihood of finding post-release defects if a file increases in size by 100 lines.

We formalize our work in the following research questions:

Q1 Which code and process metrics impact the post-release defects? Do these metrics differ for different releases of Eclipse?

Q2 By how much do the metrics impact the post-release defects? Does the level of impact change across different releases?

To evaluate our approach, we perform a case study on the Eclipse project. We were able to identify a small set (3 or 4 out of 34) of the independent variables that explain the majority of the impact on the dependent variable, post-release defects.

We also examine the predictive and explanative powers of the models built using the minimal set of independent variables. The results show that the simple logistic regression models built using our approach (i.e. using the small set of 3 or 4 independent vari-

ESEM '10, September 16-17, 2010, Bolzano-Bozen, Italy.

ables) achieve prediction and explanative results that are comparable to the more complex models that use the full set (i.e., 34) of independent variables.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 describes our approach and the data used in our study. We present the case study results and the research questions posed in Section 4 and follow with a discussion and comparison of the models built using our approach in Section 5. Section 6 presents the threats to validity and Section 7 concludes the paper.

2. RELATED WORK

The majority of the related work comes from the area of defect prediction. Most of these efforts build multivariate logistic regression models (e.g., [3–5, 10]) to predict faulty locations (e.g., files or directories). We divide the related work into two categories, based on the type of metrics used: code metrics and process metrics.

2.1 Using Code Metrics

Ohlsson and Alberg [11] use metrics that were automatically derived from design documents to predict fault-prone modules. Their set of metrics also included McCabe's cyclomatic complexity metric [12]. They performed their case study on a Ericsson Telecom software system and showed that based on design and fault data, one can build accurate prediction models, even before any coding starts. Basili *et al.* [13] used the Chidamber and Kemerer (CK) metrics suite [14] to predict class fault-proneness in 8 medium-sized information management systems. Subramanyam and Krishnan [15] performed a similar study (using the CK metrics) on a commercial system and Gyimothy *et al.* [16] performed a similar analysis on Mozilla. Their studies confirmed the findings by Basili's study. El Emam *et al.* [17] used the CK metrics, in addition to Briand's coupling metrics [18] to predict faulty classes. They reported high prediction accuracy of faulty classes using their employed metrics. Nagappan and Ball [6] used a static analysis tool to predict the pre-release defect density of the Windows Server 2003. In another study, Nagappan *et al.* [19] predicted post-release defects, at the module level, using source code metrics. They used 5 different Microsoft projects to perform their case study and found that it is possible to build prediction models for an individual project, but no single model can perform well on all projects. Zimmermann *et al.* [4] extracted an extensive set of source code metrics and used them to predict post-release defects.

The majority of the work that use code metrics to predict defects leverage multiple metrics. Some of this work recognize the possibility of multicollinearity problems, and therefore, employe PCA (e.g. [6, 19]). Although PCA may reduce the complexity of the prediction model, it does not necessarily reduce the number of metrics that need to be collected. In addition, once PCA is applied, it is difficult to directly map the independent variables to the dependent variable, since the PCs are linear combinations of many independent variables.

2.2 Using Process Metrics

Other work use process metrics, such as the number of prior defects or prior changes to predict defects. Graves *et al.* [20] showed that the number of prior changes to a file is a good predictor of defects. They also argued that change data is a better predictor of defects than code metrics in general. Studies by Arisholm and Briand [21] and Khoshgoftaar *et al.* [22] also reported that prior changes are a good predictor of defects in a file. Hassan [3] used the complexity of a code change to predict defects. He showed that prior faults is a better predictor of defects than prior changes. He then showed that using the entropy of changes is a good predictor of defects. Moser *et al.* [23] showed that process metrics perform better than code metrics, to predict post-release defects in Eclipse. They also reported that for the Eclipse project, pre-release defects seem to perform extremely well in predicting post-release defects. Yu *et al.* [24] also showed that prior defects are a good indicator of future defects.

The previous work using process metrics highlighted two metrics that seemed to perform well: prior changes and prior bug fixing changes. In our work, we annotated Zimmermann's Eclipse data set with the well-known historical predictors of defects, prior changes and prior bug fixing changes.

Our work differs from previous work, in that we focus on studying the impact of code and process metric on post-release defects, rather than predict where the post-release defects are. We use statistical techniques to identify a small, statistically significant and minimally collinear, set of the *original* metrics that impact post-release defects. We also quantify the impact of these metrics on post-release defects.

3. APPROACH

In this section, we detail the steps of our approach, shown in Figure 1. Our approach is inspired by the previous work by Cataldo *et al.* [5]. In a nutshell, our approach takes as input an extensive list of all code and process metrics (34 metrics). Then, we build a logistic regression model and analyze the statistical significance and collinearity characteristics of the independent variables (i.e., metrics) used to build the model. We eliminate the statistically insignificant and highly collinear independent variables, which leaves us with a much smaller (3 or 4 metrics) set of statistically significant and minimally collinear independent variables. The small set of metrics is then used to build a final logistic regression model, which we use to understand the impact of these metrics on post-release defects.

Each step of our approach is discussed in detail in the following subsections.

3.1 Collection and Description of Input Metrics

We use a number of metrics to study their impact on post-release defects. We acquired the latest version (i.e., 2.0a) of the publicly available Eclipse data set provided by Zimmermann *et al.* [4]. Zimmermann's data set contain a number of code metrics, as well as pre and post-release defects. We annotate the data set with the well-known process metrics: the total number of prior changes (TPC) and prior bug fixing changes (BFC).

The process metrics were extracted from the CVS [25] repository of Eclipse. We used the J-REX [26] tool, a code extractor for Java-based software systems, to extract the annotated process metrics. The J-REX tool obtains a snapshot of the Eclipse CVS repository and groups changes into transactions using a sliding window approach [27]. The CVS commit comments of the changes are examined and key words such as "bug", "fix", etc. are used to identify the bug fixing changes. A similar approach was used by Moser *et al.* [23] to classify bug fixing changes.

A total of 34 different metrics (shown in Table 2) were extracted for three different releases of Eclipse – versions 2.0, 2.1 and 3.0. All of the extracted metrics were mapped to the software locations at the file level. We list the metrics used and provide a brief description in the following subsections.

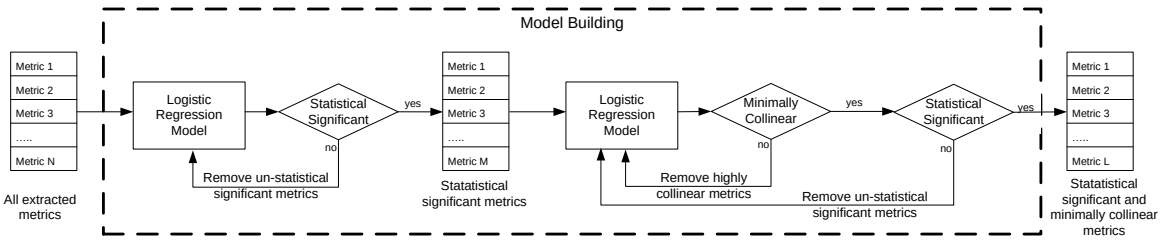

Figure 1: Approach overview

3.1.1 Process Metrics

Numerous previous studies by [3, 20, 23] show that process metrics perform well to predict software defects. In this subsection, we highlight the process metrics used in our study.

1. Total Prior Changes (TPC): Measures the total number of changes to a file in the 6 months before the release. Previous work by Moser *et al.* [23] and Graves *et al.* [20] showed that the total number of changes is a good indicator of future defects.

2. Prior Bug Fixing Changes (BFC): The number of bug fixing changes done to a file in the 6 months before the release. This metric is extracted from the CVS repository exclusively. Previous work by Yu *et al.* [24] and Hassan [3] showed that the number of previous bug fixing changes is a good indicator of future defects.

3. Pre-release defects (PRE): The number of pre-release defects in a file in the 6 months before the release. Zimmerman *et al.* used a pattern matching approach that searches for defect identification numbers in source control change comments and used the defect identifiers to classify the changes as bug fixing changes [4].

4. Post-release defects (POST): The number of post-release defects in a file in the 6 months after the release [4]. This metric is used as the dependent variable in our logistic regression models.

3.1.2 Code Metrics

An extensive set of code metrics was obtained from the Promise data set provided by Zimmermann *et al.* [4]. The majority of the metrics are complexity metrics that have been successfully used in the past [4, 23] to predict post-release defects. We list, and briefly explain the different code metrics used in our study:

1. Total Lines of Code (TLOC): Measures the total number lines of code of a file.

2. Fan out (FOUT): Measures the number of method calls of a file. Three measures are provided for FOUT, avg, max and total.

3. Method Lines of Code (MLOC): Measures number of method lines of code. Three measures are provided for MLOC, avg, max and total.

4. Nested Block Depth (NBD): Measures the nested block depth of the methods in a file. Three measures are provided for NBD, avg, max and total.

5. Number of Parameters (PAR): Measures the number of parameters of the methods in a file. Three measures are provided for PAR, avg, max and total.

6. McCabe Cyclomatic Complexity (VG): Measures the McCabe cyclomatic complexity of the methods in a file. Three measures are provided for VG, avg, max and total.

7. Number of Fields (NOF): Measures the number of fields of the classes in a file. Three measures are provided for NOF, avg, max and total.

8. Number of Methods (NOM): Measures the number of methods of the classes in a file. Three measures are provided for NOM, avg, max and total.

9. Number of Static Fields (NSF): Measures the number of static fields of the classes in a file. Three measures are provided for NSF, avg, max and total.

10. Number of Static Methods (NSM): Measures the number of static methods of the classes in a file. Three measures are provided for NSM, avg, max and total.

11. Anonymous Type Declarations (ACD): Measures the number of anonymous type declarations in a file.

12. Number of Interfaces (NOI): Measures the number of interfaces in a file.

13. Number of Classes (NOT): Measures the number of classes in a file.

3.2 Model Building

We are interested in finding out the files that are likely to have one or more post-release defects. Logistic regression models are generally used for this purpose. A logistic regression model correlates independent variables with a discrete dependent variable. In our case, the independent variables are the collection of code and process metrics and the dependent variable is a two-value variable that represents whether or not a file has one or more post-release defects. The model outputs the likelihood of a file to have one or more post-release defects.

We use the *glm* command in the R statical package [28] to build the logistic regression model. R provides us with a few tools that we can use to analyze the statistical characteristics of the model we build. We leverage these tools to study the statistical significance and collinearity attributes of the independent variables used to build the model.

Initially, we build a multivariate logistic regression model using all 34 metrics as the independent variables. Then, we perform an iterative process where we remove the statistically insignificant independent variables. Next, we perform a similar iterative process to remove highly collinear independent variables from the logistic regression model. This process is repeated until we reach a model that only contains statistically significant and minimally collinear independent variables.

3.2.1 Statistical Significance Analysis

We perform an analysis using R to study the statistical significance of each independent variable. We use the well known p-value to determine the statistical significance. Since some of the independent variables can have no statistically significant effect on the likelihood of post-release defects, including them in the prediction model may improve its overall prediction accuracy, but makes it difficult to claim that they produce the effect that we are observing.

We remove all of the independent variables that have a p-value greater than a specified threshold value. In this paper, we retained

Table 1: Example Using Eclipse 3.0 Metrics

	Iteration 1		Iteration 2			Iteration 7		Iteration 8	
Metric	**P-value**	**VIF**	**P-value**	**VIF**	...	**P-value**	**VIF**	**P-value**	**VIF**
TLOC	7.72e-05 ***	27.754974	1.36e-07 ***	13.819629	...	< 2e-16 ***	1.366347	<2e-16 ***	1.362840
PRE	< 2e-16 ***	1.319629	< 2e-16 ***	1.289907	...	< 2e-16 ***	1.275472	<2e-16 ***	1.244094
TPC	0.073345 +	2.782466	0.04729 *	2.733476	...	0.06941 +	2.698888	0.0056 **	1.093982
BFC	1.08e-05 ***	2.825389	3.93e-06 ***	2.782801	...	2.27e-06 ***	2.739084	-	-
ACD	0.087893 +	1.654916	0.00066 ***	1.437178	...	< 2e-16 ***	1.222093	0.0178 *	1.216477
FOUT avg	0.841697	46.585807	-	-	...	-	-	-	-
FOUT max	0.382543	26.176911	-	-	...	-	-	-	-
FOUT sum	0.948411	89.959363	-	-	...	-	-	-	-
MLOC avg	0.204205	105.763769	-	-	...	-	-	-	-
MLOC max	0.330231	42.238222	-	-	...	-	-	-	-
MLOC sum	0.055794 +	211.302843	0.25559	28.082202	...	-	-	-	-
NBD avg	0.092112 +	193.777683	0.09060 +	156.684860	...	-	-	-	-
NBD max	0.169100	12.477594	-	-	...	-	-	-	-
NBD sum	0.053443 +	1421.056428	0.03509 *	1206.186455	...	-	-	-	-
NOF avg	0.328731	206.270137	-	-	...	-	-	-	-
NOF max	0.229602	137.421047	-	-	...	-	-	-	-
NOF sum	0.109088	256.810067	-	-	...	-	-	-	-
NOI	0.654592	90.361904	-	-	...	-	-	-	-
NOM avg	0.138702	236.364314	-	-	...	-	-	-	-
NOM max	0.712747	154.720922	-	-	...	-	-	-	-
NOM sum	0.383255	210.771232	-	-	...	-	-	-	-
NOT	0.622797	87.947327	-	-	...	-	-	-	-
NSF avg	0.312735	61.944164	-	-	...	-	-	-	-
NSF max	0.762748	654.599118	-	-	...	-	-	-	-
NSF sum	0.582049	608.061575	-	-	...	-	-	-	-
NSM avg	0.619926	50.435605	-	-	...	-	-	-	-
NSM max	0.832793	625.599123	-	-	...	-	-	-	-
NSM sum	0.970193	544.627797	-	-	...	-	-	-	-
PAR avg	0.646235	10.135751	-	-	...	-	-	-	-
PAR max	0.003790 **	3.793562	0.41411	1.471249	...	-	-	-	-
PAR sum	0.115339	32.672658	-	-	...	-	-	-	-
VG avg	0.020032 *	376.900050	0.05768 +	310.938499	...	-	-	-	-
VG max	0.917081	23.922610	-	-	...	-	-	-	-
VG sum	0.015354 *	1928.733922	0.02532 *	1513.000445	...	-	-	-	-

(p<0.001 ***; p<0.01 **; p < 0.05 *; p<0.1 +)

all independent variables with p-value < 0.1. At the end of this step, the multivariate logistic regression model only contains independent variables that are statistically significant.

3.2.2 Collinearity Analysis

Multicollinearity can be caused by high intercorrelation between the independent variables. The problem with multicollinearity is that as the independent variables become highly correlated, it becomes more difficult to determine which independent variable is actually producing the effect on the dependent variable. In addition to making it difficult to determine the independent variable that is causing the effect, multicollinearity causes higher standard error. Therefore, it is beneficial to minimize collinearity within the independent variables of the logistic regression model.

Tolerance and Variance Inflation Factor (VIF) is often used to measure the level of multicollinearity. A tolerance value close to 1 means that there is little multicollinearity, whereas a tolerance value close to 0 indicates that multicollinearity is a threat. The VIF is the reciprocal of the tolerance. We used the `vif` command in the `Design` package for R to examine the VIF values of all independent variables used to build the multivariate logistic regression model. In this paper, we set the maximum VIF value to be 2.5, as suggested in [5].

Once we narrow down to only having statistically significant and minimally collinear independent variables, we use these variables to build the final logistic regression model.

To give an overview of the entire process, we provide an example of the multivariate logistic regression model built for Eclipse 3.0 in the next subsection.

3.3 Example

The multivariate logistic regression model used for Eclipse 3.0 is depicted in Table 1. Due to space limitations, we only include the first 2 and last 2 iterations in the table.

We start by building the model using all 34 independent variables. An examination of the model statistics reveals that only 11 of the 34 metrics are statistically significant, as shown in the iteration 1 column of Table 1. We removed the statistically insignificant independent variables and re-built the model. Once again, we examine the statistically significance of the independent vari-

Table 2: Descriptive Statistics of Examined Metrics

Metric	Mean	SD	Min	Max	Skew	Kurtosis
TLOC	123.3	233.4	3.0	4886.0	6.8	79.3
PRE	0.7	2.1	0	43	7.3	81.0
TPC	1.4	3.6	0	82.0	7.5	99.2
BFC	0.45	1.6	0	44.0	9.6	157.5
ACD	0.46	1.7	0	56.0	8.9	164.1
FOUT avg	3.0	3.6	0	60.2	3.0	22.3
FOUT max	11.2	17.0	0	334.0	5.2	55.1
FOUT sum	44.2	95.9	0	2162.0	6.3	55.1
MLOC avg	5.7	6.6	0	159.2	4.2	48.9
MLOC max	20.7	34.9	0	995.0	9.0	168.9
MLOC sum	83.9	190.5	0	4266.0	7.6	96.9
NBD avg	1.3	0.81	0	7.0	0.26	1.0
NBD max	2.4	1.9	0	17.0	0.72	0.47
NBD sum	16.9	29.7	0	621.0	5.9	62.3
NOF avg	2.5	6.3	0	355.0	24.8	1120.2
NOF max	2.9	6.7	0	355.0	20.6	848.8
NOF sum	3.1	7.1	0	355.0	18.0	681.5
NOI	0.16	0.37	0	1	1.8	1.3
NOM avg	8.5	13.3	0	284.0	7.2	90.4
NOM max	9.5	14.8	0	284.0	6.2	67.3
NOM sum	10.2	16.2	0	290.0	6.1	63.4
NOT	0.84	0.38	0	6.0	-1.4	5.1
NSF avg	2.1	18.5	0	1254.0	40.5	2287.0
NSF max	2.3	18.5	0	1254.0	39.9	2242.9
NSF sum	2.3	18.5	0	1254.0	39.9	2239.9
NSM avg	1.1	14.3	0	845.0	47.0	2443.7
NSM max	1.2	14.4	0	845.0	46.1	2380.5
NSM sum	1.2	14.4	0	845.0	46.0	2375.4
PAR avg	0.97	0.76	0	9.0	2.3	10.9
PAR max	2.3	1.8	0	30.0	2.2	13.9
PAR sum	12.1	41.7	0	2100.0	32.3	1392.9
VG avg	1.9	2.0	0	68.5	7.0	155.9
VG max	5.8	10.6	0	310.0	11.8	246.6
VG sum	28.5	61.9	0	1479.0	7.7	100.0

Table 3: Metric Correlations

Metric	PRE	TLOC	TPC	BFC
POST	0.381	0.33	0.128	0.181
TLOC	0.421	1.00	0.210	0.228
PRE	1.000	0.42	0.248	0.328
TPC	0.248	0.21	1.000	0.695
BFC	0.328	0.23	0.695	1.000
ACD	0.258	0.44	0.123	0.163
FOUT avg	0.313	0.77	0.144	0.162
FOUT max	0.375	0.87	0.185	0.203
FOUT sum	0.400	0.94	0.191	0.214
MLOC avg	0.314	0.80	0.152	0.155
MLOC max	0.380	0.90	0.185	0.195
MLOC sum	0.403	0.96	0.200	0.214
NBD avg	0.303	0.74	0.158	0.171
NBD max	0.368	0.85	0.192	0.210
NBD sum	0.392	0.95	0.197	0.219
NOF avg	0.242	0.60	0.087	0.102
NOF max	0.256	0.63	0.098	0.114
NOF sum	0.260	0.63	0.102	0.118
NOI	-0.160	-0.55	-0.022	-0.064
NOM avg	0.296	0.71	0.171	0.182
NOM max	0.314	0.74	0.184	0.196
NOM sum	0.319	0.76	0.186	0.200
NOT	0.160	0.55	0.022	0.064
NSF avg	0.174	0.33	0.079	0.109
NSF max	0.186	0.35	0.088	0.119
NSF sum	0.186	0.35	0.089	0.120
NSM avg	0.197	0.34	0.059	0.073
NSM max	0.202	0.35	0.063	0.075
NSM sum	0.202	0.35	0.063	0.075
PAR avg	0.094	0.26	0.076	0.044
PAR max	0.257	0.60	0.143	0.130
PAR sum	0.350	0.82	0.198	0.200
VG avg	0.300	0.78	0.136	0.139
VG max	0.359	0.87	0.170	0.178
VG sum	0.389	0.95	0.190	0.205

ables. This time, we observe that another 2 of the independent variables become statistically insignificant, shown in the iteration 2 column of Table 1. We continued this process of removing the statistically insignificant independent variables, rebuilding the model, re-examining the significance until all independent variables in the model were significant. This was achieved after the fourth iteration.

Then, we remove all independent variables that had a VIF value greater than 2.5. Each time a variable is removed, we made sure to check the p-values of all independent variables left in the model to assure they are still statistically significant. We did this for 4 more iterations. In the eighth iteration, we finally end up with a logistic regression model that contains 4 statistically significant and minimally collinear independent variables. The final model for the Eclipse 3.0 release only contain ACD, PRE, TPC and TLOC as independent variables.

4. CASE STUDY RESULTS

We performed a study on three different revisions of the Eclipse project. We want to examine our approach on Eclipse and identify the independent variables that produce the impact and quantify by how much they impact post-release defects. We also examine the evolution of these independent variables by building and comparing the logistic regression models for 3 different releases of Eclipse.

4.1 Preliminary Analysis of Data

Before delving into the results of our case study, we perform some preliminary analysis on the collected metrics. We calculated a few of the most common descriptive statistics, mean, min, max, standard deviation (SD) which are reported in Table 2. In addition, we calculated the skew and kurtosis measures for each metric.

Skew measures the amount of asymmetry in the probability distribution of a variable, in relation to the normal distribution. Skew can have a positive or negative value. A positive skew value indicates that the distribution is positively skewed, meaning the metric values are mostly on the low end of the scale. In contrast, a negative skew value indicates a negatively skewed distribution, where most of the metric values are on the high end of the scale. The normal distribution has a skew value of 0.

Kurtosis on the other hand characterizes the relative peakedness or flatness of a distribution, in relation to the normal distribution. A positive kurtosis value indicates a curve that is too tall and a negative kurtosis value indicates a curve that is too flat. A normal distribution has a kurtosis value of 0.

It is important to study the descriptive statistics of the metrics used to better understand the dataset at hand and perform any needed transformations. Most real world data have high to moderate skew

Table 4: VIF and P-values of Code and Process Metrics in Eclipse 2.0, 2.1 and 3.0

	Eclipse 2.0		Eclipse 2.1		Eclipse 3.0	
Metric	**P-value**	**VIF**	**P-value**	**VIF**	**P-value**	**VIF**
ACD					0.0178 *	1.216477
NSM avg			0.000266 ***	1.096400		
PAR max	5.25e-08 ***	1.289606				
PRE	< 2e-16 ***	1.178974	< 2e-16 ***	1.096400	<2e-16 ***	1.244094
TPC	< 2e-16 ***	1.084916			0.0056 **	1.093982
TLOC	< 2e-16 ***	1.392259	< 2e-16 ***	1.417422	<2e-16 ***	1.362840

(p<0.001 ***; p<0.01 **; p < 0.05 *; p<0.1 +)

and kurtosis values and transformations such as log or square root transformations are usually employed (e.g., [4, 5, 8]).

We can observe from Table 2 that most of the metrics suffer from positive skew (i.e., all the metric values are on the low scale) and have positive kurtosis values (i.e., too tall). To alleviate some of the issues caused by these higher than expected skew and kurtosis values, we log transformed all of the metrics. From this point on, whenever we mention a metric, we actually are referring to the log transformation of the metric.

In addition, Table 3 calculates the pairwise correlation measures of all the metrics against the metrics that are known to perform well in bug prediction. First, we observe that the PRE and TLOC metrics have higher correlation with POST than the change based TPC and BFC metrics. Furthermore, we observe that the TLOC metric is highly correlated with the majority of the code metrics, especially, the FOUT, MLOC, NDB, NOF and VG metrics. Similar observations were made by Graves *et al.* [20]. The high correlation values can be used as an indication of possible multicollinearity problems that may arise if these independent variables were combined in a single logistic regression model.

4.2 Identifying Code and Process Metrics that Impact Post-release Defects

Q1: Which code and process metrics impact the post-release defects? Do these metrics differ for different releases of Eclipse?

To answer this question, we followed the same steps outlined in our approach section. We build the models for the 3 different releases, Eclipse 2.0, Eclipse 2.1 and Eclipse 3.0. The results are presented in Table 4. The results indicate that using this approach, we are able to successfully build models for all 3 releases. In all 3 releases, all of the independent variables are statistically significant, with p-value < 0.1 and minimally collinear, with VIF values < 2.5. The Eclipse 2.0 and 3.0 models are composed of 4 different independent variables, while the Eclipse 2.1 model contains only 3 independent variables.

Next, we investigate whether these independent variables are the same for the different releases of Eclipse or whether they change from one release to the other. Our findings indicate that some of the independent variables change for different releases. These are mainly the code metrics (i.e., ACD for Eclipse 3.0, NSM avg for Eclipse 2.1 and PAR max for Eclipse 2.0). The TPC metric is in 2 of the 3 models, while, the TLOC and the PRE metrics were apparent in all 3 models. This finding is interesting because it shows that the simple metrics (i.e., TLOC and PRE) are actually the most stable independent variables in our model.

In the next subsection, we quantify the impact by each independent variable on the post-release defects.

Using the p-value and the VIF measures, we are able to determine which of the code and process metrics impact post-release defects. These metrics change for different releases of Eclipse.

4.3 Quantifying the Impact of Code and Process Metrics on Post-release Defects

Q2: By how much do the metrics impact the post-release defects? Does the level of impact change across different releases?

We would like to quantify the impact caused by the independent variables on post-release defects. For example, what if a file has 3 pre-release defects versus 4 pre-release defects. It would make intuitive sense that the chance of a post-release defect will increase, but by how much? Will an increase in 1 pre-release defect double the chance of a post-release defect?

To quantify the impact, we use odds ratios. Odds ratios are the exponent of the logistic regression coefficients. Odds ratios greater than 1 indicate a positive relationship between the independent and dependent variables (i.e., an increase in the independent variable will cause an increase in the likelihood of the dependent variable). Odds ratios less than 1 indicate a negative relationship, or in other words, an increase in the independent variable will cause a decrease in the likelihood of the dependent variable.

The value of the odds ratios indicate the amount of increase that 1 unit increase of the independent variable will cause to the dependent variable. Since we log all of the independent variables, 1 unit increase means a unit increase in the log-scale. We list the odds ratios of Eclipse 2.0, Eclipse 2.1 and Eclipse 3.0 in Tables 5, 6 and 7, respectively. The table lists the χ^2, p-value and deviance explained of each model. The last row of the table lists the difference of the deviance explained in percent. The models are built in a hierarchical way, meaning we build starting with 1 independent variable and keep adding the independent variables until the final model is built.

Firstly, we examine the odds ratios of Eclipse 3.0, depicted in Table 7. It can be observed that as we add metrics to the logistic regression model, the odds ratios change. This means that as we add more independent variables to the model, the impact of the individual independent variables will vary. For example, we can see that TLOC has an odds ratio of 2.25 in Table 7, model 1. This changes to 1.70 in model 4. This means that if we were *only* using the TLOC variable to build the logistic regression model, then increasing the total lines of code by 1 log unit, increases the likelihood of having a post-release defect by 125% (i.e., more than double the likelihood). On the other hand, if we combined the TLOC variable with other independent variables (as we did in model 4), then the likelihood of finding a post-release defect due to a 1 unit increase in

Table 5: Eclipse 2.0

	Model 1	Model 2	Model 3	Model 4
TLOC	2.57***	2.40***	2.11***	1.88***
TPC		1.87***	1.62***	1.62***
PRE			1.87***	1.90***
PAR max				1.73***
Model χ^2	979	1255	1375	1404
Model p-value	<0.001	<0.001	<0.001	<0.001
Deviance Explained	17.6%	22.5%	24.7%	25.2%
Model Comparison (%)	-	276 (4.9%)	120 (2.2%)	29 (0.5%)

(p<0.001 ***; p<0.01 **; p < 0.05 *; p<0.1 +)

Table 6: Logistic Regression Model Eclipse 2.1

	Model 1	Model 2	Model 3
TLOC	2.05***	1.49***	1.44***
PRE		3.27***	3.27***
NSM avg			1.21***
Model χ^2	605	944	957
Model p-value	<0.001	<0.001	<0.001
Deviance Explained	11.2%	17.5%	17.7%
Model Comparison (%)	-	339 (6.3%)	13 (0.2%)

(p<0.001 ***; p<0.01 **; p < 0.05 *; p<0.1 +)

Table 7: Eclipse 3.0

	Model 1	Model 2	Model 3	Model 4
TLOC	2.25***	1.30***	1.66***	1.70***
TPC		2.15***	1.10**	1.11**
PRE			3.24***	3.28***
ACD				0.87*
Model χ^2	1289	1347	1877	1883
Model p-value	<0.001	<0.001	<0.001	<0.001
Deviance Explained	14.5%	15.2%	21.1%	21.2%
Model Comparison (%)	-	58 (0.7%)	530 (5.9%)	6 (0.1%)

(p<0.001 ***; p<0.01 **; p < 0.05 *; p<0.1 +)

TLOC is only 70%. However, we can see from the same table that as we add more of the independent variables to the model, the deviance explained and the χ^2 value increase significantly, indicating a significant improvement in the explanative power of the model.

Using model 4 in Table 7, we can see that for TLOC , TPC and PRE the odds ratios suggest an increase in the likelihood of a post-release defect. On the other hand, a log unit increase in number of anonymous declaration types (ACD) produces a negative impact on the likelihood of finding one or more post-release defects. In fact, for every unit increase in ACD, the chance of finding a post-release defect decreases by 13%.

We can also use the odds ratios value to examine the impact on post-release defects if a file was to increase by 100 lines. To do so, we exponentiate the odds ratio value for that independent variable by the quantity increase in units. For example, if TLOC was increased by 100 lines, and since we are using log this would be 2 units (i.e. log(100) = 2)). Therefore, the impact of a 100 line increase, using model 4 in Table 7, is $1.70^2 = 2.89$. This means that if we increase the TLOC by 100 lines, then the chance of a post-release defect is increased by 189%.

Comparing the odds ratios of the independent variables for different releases, we can see that the odds ratios change slightly. At the same time, we can observe that they are stable, meaning, if they are above 1 for one of the releases, they stay that way for all the other releases. For example, the odds ratio value for TLOC is positively associated with post-release defects in all 3 releases. This finding was also observed by Briand et al. [29]. A similar observation holds for the PRE metric as well. Studying the stability is important because the stability of a metric tells us whether we can draw conclusions about the impact produced by the metric for this project.

Table 8: Comparison of Precision, Recall and Accuracy of Prediction Models

	Eclipse 2.0		Eclipse 2.1		Eclipse 3.0	
	Ours	**All metrics**	**Ours**	**All metrics**	**Ours**	**All metrics**
Precision (%)	66.3	63.6	60.0	58.6	64.1	64.7
Recall (%)	28.5	32.4	15.8	17.2	25.7	26.5
Accuracy (%)	87.5	87.5	89.8	89.8	86.9	87.0

We are able to quantify the impact produced by the code and process metrics on post-release defects using odds ratios. The impact on post-release defects changes for different releases of Eclipse.

5. DISCUSSION

The main goal of our study is to minimize the large number of independent variables in the multivariate logistic regression model, in order to better understanding the impact of the various independent variables on the dependent variable, post-release defects. In the previous section, we performed a case study where our initial set of metrics is 34 and we were able to reduce that number to 3 or 4 statistically significant and minimally collinear set of metrics.

Although we were successful in using our approach to understand the impact of the process and code metrics on post-release defects, a few questions still linger: *How does our approach affect the prediction accuracy of the model and how does it compare to previous techniques used to deal with the issue of multicollinearity, namely PCA?* We answer these questions in the following subsections.

5.1 Comparing Prediction Accuracy

To study the prediction accuracy, we build two multivariate logistic regression models: one that uses all of the metrics and one that uses the smaller set of statistically and minimally collinear metrics. The logistic regression models predict the likelihood of a file being defect prone or not. The output of the model is given as a value between 0 and 1. We classified files with predicted likelihoods above 0.5 as defect prone, otherwise they are classified as being defect free.

The classification results of the prediction models were stored in a confusion matrix, as shown in Table 9.

Table 9: Confusion matrix

		True Class	
		Defect	No Defect
Predicted	Defect	a	b
	No Defect	c	d

The performance of the prediction model is measured using three different measures:

1. **Precision:** Relates number of files predicted *and* observed as defect prone to the number of files predicted as defect prone. It is calculated as $\frac{a}{a+b}$.
2. **Recall:** Relates number of files predicted *and* observed as defect prone to the number of files that actually had defects. It is calculated as $\frac{a}{a+c}$.
3. **Accuracy:** Relates the total number of correctly classified files to the total number of files. It is calculated as $\frac{a+d}{a+b+c+d}$.

The prediction results for all 3 release of Eclipse are presented in Table 8. The results in the table agree with previous results obtained by Zimmermann *et al.* in [4]. In all releases, our results were very close to those generated by the model that uses all 34 metrics. It is important to highlight that the main goal of our approach is not to achieve better prediction results. Our main goal is to understand the impact of the independent variables on post-release defects. We proved that we are able to study the impact without significantly affecting the prediction accuracy of the models. Furthermore, it is important to note here that our models are far less complex, using only 3 or 4 metrics.

Using a much smaller set of statistically significant and minimally collinear set of metrics does not significantly affect the prediction results of the logistic regression model.

5.2 Comparing with Principle Component Analysis

Multicollinearity is caused by using highly correlated independent variables, which makes it more and more difficult to determine which one of the independent variables is producing the effect on the dependent variable. Previous research (e.g., [6, 7, 10, 19]) addressed the multicollinearity problem by employing Principal Component Analysis (PCA) [30]. PCA uses the original metrics to build Principal Components (PCs) that are orthogonal to each other. The PCs are linear combinations of the metrics. These PCs are then used as the independent variables in the logistic regression model.

Although using PCA solves the issue of multicollinearity, it has its disadvantages as well. First, PCA does not necessarily reduce the number of independent variables, since each PC is a linear combination of all the input metrics. For this reason, models that use PCA may still need to collect many input metrics. Second, once the PCs are used to predict defects, it is very difficult to pin-point which of the original metrics (used to build the PCs) actually produced the effect. Not being able to pin-point which of the input metrics actually caused the effect is a major disadvantage. It makes it much harder for practitioners and managers to understand the prediction models, causing them to disregard the models or search somewhere else for answers.

In this section, we compare the results of the models generated using our approach to models that we build using PCA. We perform this comparison to verify the validity of our approach and measure its performance in comparison to models that would be built with all 34 metrics.

To build the PCA models, we input all of the independent variables and build the PCs. Then, we measure the % cumulative variation when a different number of PCs is used. Based on the % of cumulative variation we wish to achieve, we use a different number of PCs as input to the logistic regression model. For example, in Eclipse 3.0, to achieve a % cumulative variation of 95%, we would require a minimum of 8 PCs. To achieve a 99% cumulative

Table 10: Comparing our approach to PCA

	Eclipse 2.0				Eclipse 2.1				Eclipse 3.0			
Cumulative Variability	**Ours**	**95%**	**99%**	**100%**	**Ours**	**95%**	**99%**	**100%**	**Ours**	**95%**	**99%**	**100%**
Model χ^2	1404	1375	1487	1552	957	710	981	979	1886	1451	1924	1953
No. of metrics	4	34	34	34	3	34	34	34	4	33	33	33
Min. no. of PCs	-	7	15	32	-	7	15	33	-	8	15	33
Model p-value	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
Deviance Explained	25.2%	24.7%	26.7%	27.9%	17.7%	13.1%	18.1%	18.1%	21.2%	16.3%	21.7%	22.0%
Comparison (%)	-	+0.5%	-1.5%	-2.7%	-	+4.6%	-0.4%	-0.4%	-	+4.9%	-0.5%	-0.8%

variation, we require a minimum of 15 PCs. A 95% cumulative variation was commonly used in the previous work on defect prediction [6, 7, 10, 19, 31].

The main reason previous work used 95% cumulative variation was to reduce the data needed to build the models. For example, for Eclipse 3.0 using 8 PCs instead of 34 PCs, converts to a data reduction of 76.5%. However, as we will show, this does not necessarily mean that less metrics need to collected, as the number of metrics used for Eclipse 3.0 is 33 out of 34. This is a data reduction of 2.9%. For the sake of comparison, we compare the models generated using our approach, to PCA-based models that can achieve 95%, 99% and 100% cumulative variation.

To compare, we use 4 different measures. First, we measure the χ^2 value achieved by the different models. In addition, we report the percentage of deviance explained for each model to examine the explanative power of the models. Furthermore, we record the number of PCs required to achieve the various levels of cumulative proportions of variance. The number of independent variables required to build the PCA-based models is also reported. Finally, we calculate the p-value of the models generated to make sure that the models are statistically significant.

Our comparison is reported in Table 10. The last row of the table represents the difference in deviance explained comparing to our model. A positive value means our model outperforms the PCA-based model, and vice versa. In all 3 releases, our model can outperform the PCA-based models with 95% cumulative probability of variance in terms of χ^2 value and deviance explained. As we stated earlier, most of the previous work (e.g., [6,7,10,19,31]), used the PCA-based models with 95% cumulative variation. Furthermore, we can see that our model uses far less metrics than the PCA models. To calculate the number of metrics required for the PCA-models, we examined the 'loadings' of the PCs. Hair *et al.* [32] suggested that loading values below 0.4 are considered to have a low rank in the PCs. We counted the number of metrics that have a loading value greater than 0.4. As shown in Table 10, this number was 34 metrics for release 2.0, 34 metrics for release 2.1 and 33 metrics for release 3.0. The only metric that did not have a loading value greater than 0.4 is the VG sum metric in Eclipse 3.0.

Finally, we would like to point out two main advantages of our models. First, our models require far less metrics, meaning that there is a significant amount of savings in the effort that needs to be put into the extraction of these metrics. Second, and most importantly, our models are simple and explainable. They can be used by practitioners to *understand* the impact of the independent variables on post-release defects. This is not easily achieved with models that use PCA.

Using our approach, we are able to build logistic regression models that can be used to understand the impact of the code and process metrics on post-release defects. Our models can achieve better explanative power than PCA-based models that explain 95% cumulative variation.

6. THREATS TO VALIDITY

Our analysis is based on 3 different releases of the Eclipse project. Although the Eclipse project is a large open source project, our results may not generalize to other projects. In addition, the list of metrics used in our study to build the logistic regression models is by no means complete. Therefore, using other metrics may yield different results. However, we believe that the same approach can be applied on any list of metrics. The VIF and p-value thresholds used in our study were chosen because they proved to be successful in previous studies [5]. Other cutoff values may be used for the VIF and p-value and may yield slightly different results.

7. CONCLUSION

A large amount of effort has been put into prediction models that aim to find the locations (i.e., files or folders) of defects in a software system. As this area of research grows, a greater number of metrics is being used to predict defects. This increase in metrics increases the complexity of the prediction models, decreasing the chance of their adoption in practice. In addition, increasing the number of metrics increases the chance of multicollinearity, which makes it difficult to determine which of these metrics *actually* impact post-release defects.

In this paper, we put forth an approach that reduces the number of metrics to a much smaller, statistically significant and minimally collinear set. The small set of metrics are then used to build logistic regression models. We use odds ratios to quantify the impact of the various independent variables (i.e., code and process metrics) on the dependent variable, post-release defects.

Finally, we compared the prediction accuracy of the models built using our approach to models that use the full set of metrics. We found very little difference in the prediction accuracy, yet our models used significantly less metrics. We also compared the explanative power of the logistic regression models using our approach and found that models built using our approach can outperform PCA-based models that explain 95% cumulative variation, and perform within 2.7% of PCA-based models that explain 100% cumulative variation.

8. REFERENCES

[1] L. Erlikh, "Leveraging legacy system dollars for e-business," *IT Professional*, vol. 2, no. 3, pp. 17–23, 2000.

[2] J. Moad, "Maintaining the competitive edge," *Datamation*, vol. 64, no. 66, pp. 61–62, 1990.

[3] A. E. Hassan, "Predicting faults using the complexity of code changes," in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88.

[4] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for Eclipse," in *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, 2007, p. 9.

[5] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *IEEE Transactions on Software Engineering*, vol. 99, no. 6, pp. 864–878, 2009.

[6] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005, pp. 580–586.

[7] ——, "Use of relative code churn measures to predict system defect density," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005, pp. 284–292.

[8] Y. Shin, R. Bell, T. Ostrand, and E. Weyuker, "Does calling structure information improve the accuracy of fault prediction?" *International Working Conference on Mining Software Repositories,*, pp. 61–70, 2009.

[9] H. Motulsky, "Multicollinearity in multiple regression," Online: http://www.graphpad.com/articles/Multicollinearity.htm, Accessed March 2010.

[10] M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," *Working Conference on Reverse Engineering*, pp. 135–144, 2009.

[11] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Trans. Softw. Eng.*, vol. 22, no. 12, pp. 886–894, 1996.

[12] T. J. McCabe, "A complexity measure," in *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, 1976, p. 407.

[13] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, 1996.

[14] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[15] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *IEEE Trans. Softw. Eng.*, vol. 29, no. 4, pp. 297–310, 2003.

[16] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 897–910, 2005.

[17] K. E. Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *J. Syst. Softw.*, vol. 56, no. 1, pp. 63–75, 2001.

[18] L. C. Briand, J. W. Daly, and J. K. Wüst, "A unified framework for coupling measurement in object-oriented systems," *IEEE Trans. Softw. Eng.*, vol. 25, no. 1, pp. 91–121, 1999.

[19] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 452–461.

[20] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions of Software Engineering*, vol. 26, no. 7, pp. 653–661, July 2000.

[21] E. Arisholm and L. C. Briand, "Predicting fault-prone components in a Java legacy system," in *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, 2006, pp. 8–17.

[22] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl, "Data mining for predictors of software quality," *International Journal of Software Engineering and Knowledge Engineering*, vol. 9, no. 5, pp. 547–564, 1999.

[23] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*, 2008, pp. 181–190.

[24] T.-J. Yu, V. Y. Shen, and H. E. Dunsmore, "An analysis of several software defect models," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, pp. 1261–1270, 1988.

[25] W. F. Tichy, "RCS - a system for version control," *Software: Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985.

[26] W. Shang, Z. M. Jiang, B. Adams, and A. E. Hassan, "Mapreduce as a general framework to support research in mining software repositories (MSR)," in *MSR '09: Proceedings of the Fourth International Workshop on Mining Software Repositories*, 2009, p. 10.

[27] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, 2005, pp. 1–5.

[28] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, 2009. [Online]. Available: http://www.R-project.org

[29] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter, "Exploring the relationship between design measures and software quality in object-oriented systems," *J. Syst. Softw.*, vol. 51, no. 3, pp. 245–273, 2000.

[30] I. T. Jolliffe, *Principal Component Analysis*, 2nd ed. Springer, October 2002.

[31] G. Denaro and M. Pezzè, "An empirical evaluation of fault-proneness models," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, 2002, pp. 241–251.

[32] J. F. Hair, Jr., R. E. Anderson, and R. L. Tatham, *Multivariate data analysis with readings (2nd ed.)*. Macmillan Publishing Co., Inc., 1986.