Software Defect Prediction from Code Quality Measurements via Machine Learning

by

Ross Earle MacDonald

Thesis submitted to Saint Mary's University in partial fulfilment of the requirements for the Degree of Master of Science in Computing and Data Analytics

August, 2018, Halifax, Nova Scotia

Approved by: Dr. Pawan Lingras
Supervisor
Department of Mathematics
and Computing Science

Approved by: Dr. Stavros Konstantinidis
Supervisory Committee Member
Department of Mathematics
and Computing Science

Approved by: Dr. Yasushi Akiyama
Supervisory Committee Member
Department of Mathematics
and Computing Science

Approved by: Dr. Daniel L. Silver
External Examiner
Acadia University

Abstract

Software Defect Prediction from Code Quality Measurements via Machine Learning

by Ross Earle MacDonald

Improvement in software development practices to predict and reduce software defects can lead to major cost savings. The goal of this thesis is to demonstrate the value of static analysis metrics and rules in predicting software defects at a much larger scale than previous efforts. The study analyses data collected from more than 500 software applications, across 3 multi-year software development programs, and uses over 150 software static analysis measurements. Static analysis metrics, rule violations and software defect historical actual values are sourced from multiple disparate databases, joined and groomed for analysis. Several feature selection techniques are employed to narrow the feature set focus to the most influential variables. Furthermore, a number of machine learning techniques such as neural network and random forest are used to determine whether seemingly innocuous rule violations can be used as significant predictors of software defect rates.

August, 2018, Halifax, Nova Scotia

Software Defect Prediction from Code Quality Measurements via Machine Learning

by

Ross Earle MacDonald

Thesis submitted to Saint Mary's University in partial fulfilment of the requirements for the Degree of Master of Science in Computing and Data Analytics

August, 2018, Halifax, Nova Scotia

Approved by: Dr. Pawan Lingras
Supervisor
Department of Mathematics
and Computing Science

Approved by: Dr. Stavros Konstantinidis
Supervisory Committee Member
Department of Mathematics
and Computing Science

Approved by: Dr. Yasushi Akiyama
Supervisory Committee Member
Department of Mathematics
and Computing Science

Approved by: Dr. Daniel L. Silver
External Examiner
Acadia University

Contents

List of Figures

List of Tables

1 Introduction

1.1 Problem Statement

Software development projects are becoming increasingly large, complex, expensive, and difficult to predict [1]. One of the many issues with this escalation of complexity is the increased risk associated with cost or schedule overruns in software projects. A major contributor to cost and schedule impact in development programs is the detection of latent defects in software [2]. If the number of defects in a development project could be better anticipated, then programs would be much less prone to budget or schedule overruns and the potential for mitigation strategies could be increased. Such mitigation strategies could include: focused code reviews, increased unit testing of particularly problematic code, and added supervision of junior developers, to name a few.

Static analysis tools such as SonarQube provide key insight into the quality of code much sooner than would otherwise be obtained if a development team were to wait for the code to be tested manually [3]. The tool performs static analysis of the code prior to compilation and execution, and because of this, the tool has greater insight into the inner workings of the software than a tester may have [4]. Unfortunately, tools such as SonarQube provide so many metrics and rule count violation reports that it can be difficult to focus on the important metrics and rules, potentially overwhelming a software development team [5]. Due to this problem and the need to better predict software defects, this research will attempt to address these issues through a

comprehensive analysis of a large dataset.

This study analyses data collected from more than 500 software applications, across 3 multi-year software development programs, and uses over 150 software static analysis metrics and rules collected and provided by the SonarQube application. The development programs analysed in this research were composed entirely of modern, object oriented, Java-based software development for mission-critical and safety-critical applications. To our knowledge, no research to date has used such a large and modern data set for this purpose, and the goal is to arrive at meaningful and time-appropriate predictions, conclusions and recommendations.

1.2 State of the Art

While recent research has demonstrated promising results of Machine learning being used to predict software defect rates based on Static Analysis metrics such as Code Complexity and Code Size [6, 7], the industry-standard is still languishing behind. Most software development projects still use archaic defect prediction methods, much of which are focused on the relationship between the number of lines of code with the number of defects in an application. While a significant correlation between these two measurements has been proven a number of times [8], the implication of this relationship is uninformative.

Why industries do not adopt more innovative and informative measurement-driven prediction can be attributed to many factors, one of which is the

issue of Metrics Galore [5]. This situation can arise when a development team is presented with an overwhelmingly large number of metrics and rules to analyse. One such outcome of this situation is for the team to simply continue using the status quo: Lines of code.

Clearly, a new solution is required to narrow the focus of software development teams to a small list of important measurements that are significantly related to software defect prediction.

1.3 Thesis Objectives

The ultimate goal of this study is to demonstrate the value of static analysis metrics and rules in predicting software defects and provide development teams with a manageable list of the most significant metrics and rules. While it is difficult to define a specific threshold for avoiding the issues of Metrics Galore [5], for the purposes of this research, an arbitrary number of 20 static analysis metrics and rules will be used as the cut-off for team focus. It is expected that through this goal, future software development projects will be more successful, encounter less defects and be easier to predict.

More specifically, the fundamental objectives of this thesis are to:

1. Demonstrate a statistically significant correlation between static analysis metrics and rules and the corresponding software defects predictions.

2. Produce a machine learning model that can be used to accurately predict software defects. Given an RMSE lower than the mean distribution of 40 or an AUC greater than 0.683 [7]. See section 2.7 for further

details on why these values were chosen.

3. Identify the 10 most significant static analysis metrics and rules that attribute to the prediction of the software defects.

4. Provide an assessment of the results and recommendations for future software development projects.

1.4 Thesis Overview

In order to accomplish the objectives of this analysis, a thorough understanding of the subject matter is required. This is accomplished through an extensive literature review. The material that is presented in this thesis spans all relevant topics including:

- Software Development methodologies and current issues.
- Software Defect background and implications.
- Current software defect prediction methods.
- Software Static Analysis theory and developments.
- Machine learning background.
- Methods for predicting software defects using machine learning.

Once the subject matter is thoroughly assessed, a clear methodology is needed. This section of the thesis details the steps that have been taken in order to prepare, process, train, test, and interpret the software defect

and static analysis data. The methodology will outline the acceptance criteria for the results as well as any particular analysis methods used.

After the methodology is clearly laid out, the bulk of the analysis can be performed. The analysis accomplished in this research is documented in the results section of this thesis. All relevant model performance results will be provided and a brief discussion will be included for each prediction type and model.

A comprehensive discussion follows the results section of this thesis. The discussion includes an analysis of the findings, performance of the model and a logical analysis of the findings in order to interpret their meaning. Using the results and interpretations, some conclusions are then presented.

Finally, the thesis will include a summary section which will provide a recap of the analysis performed as well as an overall set of conclusions and some recommendations on future research topics.

2 Literature Review

2.1 Software Development

Lehman's Laws of Software Evolution have demonstrated that software development programs are becoming larger and more complex as the years go by [1]. With this ever-increasing complexity and size and decreasing quality, it is becoming increasingly difficult to accurately predict how a development program will unfold. As discussed by Brooks [9], software management teams often search for the "silver bullet" solution to this problem. Unfortunately there is no single solution that could even offer an order of magnitude improvement to performance, instead, software development teams must work to improve their efficiency and quality through many facets [9].

Software development companies require insight into the amount of labour required in order to complete a development program. By calculating this labour, the company is able to estimate the cost, schedule and resources required to complete the program within an allotted time period [10]. Software defects are one of the most important, but also one of the most difficult aspects of software development estimation [11]. Due to this difficulty, much research has been invested into this field, namely the formalization of the software development process.

Software development life-cycles have undergone various changes through the years. In the early development years, most large development programs were run in what is called a "Waterfall Process" [12, 13]. This process clearly

defines discrete states of software development, typically as follows:

1. Requirements Analysis
2. Detailed Design
3. Software Development
4. Software Integration
5. Software Testing
6. Product Acceptance Testing

The Waterfall Process breaks a software development program down into these discrete steps where the inputs, work, and outputs are clearly defined. The Requirements Analysis section involves reviewing the customer requirements and producing a set of documents that describe the technical interpretation of the customer requirements at a much finer level of detail. Once this detail is available, the team can then perform the Detailed Design analysis, which typically involves the production of UML diagrams [14] that are used to describe the behaviour of the system including internal and external sub-system interactions. Next, software development can commence, which involves developers consuming the requirements and design documentation and using it as blueprints to construct the software. Once the software is completed, it can be integrated together and executed in a test environment. Once functioning together as a cohesive unit, the software can be tested, typically first against the work products produced in the requirements analysis phase of the program. Once those requirements are verified, the software can then be acceptance tested against the customer requirements, often with the

customer involved in order to demonstrate compliance [15].

In many notable examples, large software development programs that followed the Waterfall Process encountered many significant issues. The 1994 Standish Chaos Report [16] indicates that only 16.2% of software programs that they surveyed had succeeded without significant issues. Notable failures include the 1987 California DMV Software Update program which was cancelled after $45M of investment [16]. Additionally, American Airlines invested $165M into a car rental and hotel reservation system before it was cancelled [16]. Many software development experts [12] claimed that following such rigid processes could have led to the failures witnessed in these examples. Others add that by deferring testing to the end of a program meant that the cost of fixing the software defects discovered in the test phase were significantly higher than if they had been discovered sooner [13].

Over time, software development practices changed, with the introduction of the Agile Manifesto [17]. Since then, additional specializations of Agile and iterative processes were developed such as Scrum [18], Kanban [19], Extreme Programming [20] as well as many others. These development practices encouraged breaking the software development problem down into small iterations of work, in some cases, resembling that of the Waterfall Process, but on a much smaller scale. In terms of Scrum, the development team is responsible for performing short iterations of development, each of which includes a definition phase, a design phase, an implementation phase and then a testing and verification phase [18]. Significantly fewer development programs have demonstrated major issues when using Agile methodologies [21].

It has been shown that Agile software development is more adaptable to changing requirements and complexities of software programs, which can accommodate the ever increasing size of new software development undertakings [21]. However, most complex software development programs are still prone to issues despite the relative improvements that Agile processes have brought forward [22]. It has been demonstrated that different types of Agile methodologies apply better to different situations. For example, Scrum is ideally suited to development programs where much of the specific detail is not known at the beginning of the program [23]. Aside from these differences, studies have shown that the area of software development that requires additional attention is within testing and verification [22].

As Oberscheven [24] explores, the collection and monitoring of software characteristic measurements, also called metrics, must be incorporated into an Agile methodology in order to improve the success rate of such development programs. More specifically, it is suggested that Agile software development teams perform metrics analysis and review at the end of each development iteration. This applies quite well to the Scrum concept of a retrospective [18], in that the team reviews their development metrics and they collectively make an agreement to focus on improving a measurement that is unfavourable. The team then regularly reviews these key metrics at the end of each iteration, monitoring and adjusting their strategy depending on how the measurement changes [24].

2.2 Software Defects

Software defects can cause various issues both during and after the main phase of software development has completed. Software defects present in code during development can hinder the progress of developers in completing their task and cause the development program to absorb unplanned expenses. Likewise, defects that are found during testing can be difficult and cost significant effort to rectify. Worse still, depending on the severity of the defect and the criticality of the software product, the defect could also endanger lives [2].

If more software defects are found during testing than expected, not only can this cost the company money, but it can also impair their ability to meet customer deadlines or mission milestones. Software defects can also complicate the warranty period of a software product, thus making it difficult for a software development company to move on from a program when it needs to continue to dedicate resources to supporting customers as they find new defects through normal use [25].

It has been demonstrated in many reviews that the later a software defect is found during development, the more costly it becomes to fix [26]. One reason for this escalation has been attributed to the fact that the longer a development program runs, the higher the overhead cost and the higher the cost required to change the system [26]. For example, it has been shown that a defect found during the maintenance period of an application can cost 100-times that of the cost to fix it during the main software development

phase of the program [27]. The aforementioned research demonstrates that it is imperative to predict defect rates and detect such defects as soon as possible in order to reduce software development cost.

2.3 Current State of Defect Prediction Methods

Generalized software development performance measurements have been tested against the rate of defects in software applications. For example, one study [28] demonstrated a strong correlation between requirements quality, design quality and software defects. The study showed that there was a negative correlation between the number of requirements and the number of defects found in the software - this is to say that insufficient numbers of requirements cause higher defect rates. Conversely, it was found that developing fewer design artefacts resulted in higher defect rates - this is to say that developers writing code that was not thoroughly designed will result in higher defect rates.

Another study found a significant correlation between the defect rate reported by a static lint tool with the actual software defect rate witnessed in the software application [29]. The research demonstrated that if the lint tool found a large number of errors in a particular software application, then it was highly likely that software application would contain a large number of latent defects.

Similarly, others found correlations between the number of lines of code of a software application with the number of defects detected on that software ap-

plication [8]. This is a seemingly obvious conclusion: The larger the software application, the higher the defect rate of that application. The research proposed that the extraction of more granular measurements of software quality could be used by a machine learning algorithm to predict defect rates because of the nature of the data. It also hypothesized that because the data can be easily categorized, it would be very suitable for a classification-type analysis.

2.4 Static Analysis

Static Analysis originated with the development of the Lint application by Johnson in the 1970s [30]. The Lint application was intended to remove unwanted parts of code that could cause issues. The name is derived from the same term used to describe unwanted fragments of hair in sheep wool. Johnson set the stage for the formal definition of static analysis tools by describing the difference between Lint and Compilers: "Lint takes a more global, leisurely view of the program, looking much more carefully at the compatibilities." [30, p. 1].

Fairley [4] provided a more formal definition for static analysers versus dynamic testing: static analysers look at the code without actually compiling or executing it while dynamic testing requires the application to be run and exercised. While dynamic testing is certainly required, it can be costly to perform repeatedly and can also suffer from reproducibility problems and completeness as human error is often involved.

Studies have demonstrated correlations with some static analysis measurements and software defects [31]. The most commonly used measurements for software defect predictions are software size and software complexity. Unfortunately, it has been demonstrated that although size and complexity can be strongly correlated with software defect outcomes, these measurements are self-fulfilling and not easy to control. Basic logic can be applied to derive the same results: It seems obvious that an application that is larger than another would have the potential to contain more software defects than the other. Similarly, an application that is more complex than another could also contain more defects. The issue lies in the fact that a software development team can do very little to affect these measurements: given consistent developer experience, a software application will be as large and as complex as it needs to be to solve the problem [31].

Many static analysis tools exist today, each of which provides a particular focus on the quality of software. The problem is choosing the tool or tools for a given software development program is complex and depends on many factors [32]. It has been demonstrated by comparing several static analysis tools, that the PMD Source Code Analyzer (PMD) is a very thorough Static Analysis tool, but often generates a lot of false positives and can take a long time to execute analysis. Conversely, FindBugs completes analysis quickly, returns very little false positives, but can occasionally miss an important defect in some complex code. It is also shown that although there is some overlap between tools, each tool also detects unique errors that the other tools do not. The research goes on to recommend that software development

teams should utilize a "meta-tool" that would allow them to combine the results from various static analysis tools together, thus achieving a better results. [32]

Several tools have been recently introduced that accomplish the above suggestions [32]. SonarQube is such a "meta-tool" that provides a software development team with the ability to import, customize and automatically execute static analysis on code utilizing a variety of rule definitions. Campbell and colleagues [3] describe in great detail level of customization and capability of SonarQube. For example, SonarQube can be configured to execute analysis using FindBugs rules, JLint rules as well as stock SonarQube rules and metrics on each software module. Additionally, SonarQube can be configured to execute rule analysis with customized user rules in order to fit specialized software needs.

Unfortunately, with the advent of meta-tools such as SonarQube, a new problem is created: Metrics Galore [5]. Metrics Galore is a situation where a team is paralysed by attempting to monitor too many metrics simultaneously. Many issues can occur such as de-motivation of the team, focusing on the wrong metrics, losing sight of the important goals of the program, etc. Thus it is necessary for program managers to use some means to narrow their focus on the most important metrics that affect the software quality.

In order to address the Metrics Galore issue, while still providing metrics and rules for the software development team to track and improve upon, it is possible to employ machine learning algorithms to help. While unsupervised learning can provide a solution to the overwhelming metrics issue [33], regres-

sion and classification could serve to solve the prediction issues by providing a predictive model that could be used by the development team and by the software development managers [32].

2.5 Machine Learning

Machine Learning involves training a computer model with data or historical information in an attempt to optimize a system configuration [34], provide insight into the behaviour of the system and to potentially predict the behaviour of the system in the future. Machine Learning can be divided into three main subsets:

1. Reinforcement Learning
2. Supervised Learning
3. Unsupervised Learning

2.5.1 Reinforcement Learning

Reinforcement learning is used in live systems with a feedback mechanism [34]. This form of learning involves a method of providing information continuously to the model, allowing the model to make decisions and then either reinforcing correct ones or correcting incorrect ones. The longer the model runs, presumably, the more accurate the model becomes. This method of machine learning differs from most in that there are no specific outcomes fed into the model. Instead, the model uses a reward function that is com-

puted after taking a set of actions. The actions then provide the model with a certain amount of reward. The model then reworks its behavior in order to maximize the reward earned by its actions [34]. Other research demonstrates that an excellent application of Reinforcement learning is in training decentralized robotic behaviour [35].

2.5.2 Supervised Learning

Supervised learning involves the use of historical predictors and outcomes with the intent that the model will provide useful predictions of new values given new combinations of predictors [34]. This learning is "supervised" in that the outcomes of certain sets of predictors are already known and can be used to monitor the accuracy of the predictions that the model produces.

Supervised learning algorithms come in many forms with specific strengths, weaknesses and purposes [36]. Specific models that are suitable for the research in this thesis include: Linear Regression, Decision Trees, Support Vector Machines (SVM), Neural Networks, and Random Forests. Each of these models can provide insight into the relationship between a set of features and the corresponding outcomes in a manner that permits the prediction of new values given novel feature inputs [37].

Linear regression models can be used for performing standard regression for a series of outcomes given a particular set of inputs, or features. A linear regression model is especially useful when the relationship between

the features and the outcomes is linear in nature, this is to say that the true mean of the outcome varies linearly with the features [38]. Where linear regression models fail is when the relationship is non-linear as well as when too many features are used to fit to the model. In the former case, either the data must be transformed into the linear domain, or a different model must be used. In the latter case, adding more features to a model may decrease the error on the training set, but the test set may exhibit an increase of error, this is known as an over-fitted model [36].

The classical implementation of Decision Trees is limited in application to classification analysis only. They work by dividing the data space up into consecutively smaller sections, each of which is fit by a simple model, such as a constant. This process is repeated recursively until the model hits a predetermined splitting limit or tree maximum imposed by the modeller [39]. Decision Tree splitting is typically done based on a threshold of a feature leading to the outcome. This will look like a simple "if, then, else" statement where the tree branches if a feature value is equal to or greater than a specific value. The tree branches at this point, and additional feature values are analyzed, eventually leading to an outcome decision. [37]. Decision trees are especially good at handling data that contains missing or noisy values. Conversely, Decision Trees can suffer from high error rates when the input data set is small, more so than other machine learning methods, and thus Decision Trees are more effective on larger data sets [36].

SVMs work by transforming the input feature space such that in the transformed frame of reference, the outcomes are linearly separable [36]. Support

vectors are used to accomplish this transformation, and the formulas used in SVM models require iteration over tuning parameters in order to find the best fit. SVM models have demonstrated good performance when classifying data for image recognition as well as image to text conversions [40]. SVMs come with the added complexity of several tuning parameters and may require many executions with permutations in the tuning parameters in order to find a good fit, which could be problematic for large data sets [36].

Basic Neural Networks are non-linear regression and classification modelling algorithms. Their basis is on a series of neurons which invoke the sigmoid function against a weight and input parameter, thus producing a 1 or a 0 for an output. With a large number of these neurons arranged in layers, it is possible to train Neural Networks to produce accurate predictions for a wide range of data sets [41]. However, they are especially sensitive to over-fitting, and thus care must be taken to properly perform cross-validation on any results produced by a Neural Network [36].

Random Forests work by building a set of Decision Trees based on random samples of the original data set, using replacement, also known as bootstrap aggregation, repeatedly over an entire forest possible of Decision Trees. By performing this aggregation repeatedly, iteratively and randomly many times, Random Forests minimize the noisy results of a typical Decision Tree while maintaining the benefits of a tree structure [36]. In recent years, Random Forests have become quite popular as they have demonstrated significantly higher performance on various classification problems over other models and it is hypothesized that this is mainly due to the Law of Large Numbers [42]. It

should be pointed out, however, that Random Forests can be computationally intensive as well, and do not perform as well in regression as they do in classification [42].

2.5.3 Unsupervised Learning

Unsupervised learning is used when the order of outcomes of a dataset are unknown and the user is looking for a pattern to analyse [34]. This is especially useful in situations where the distribution of the data is unknown and the researcher is looking for some additional information about the behaviour of the data-space [34]. No expectations of results are fed into the system by the analyst, instead, unsupervised learning models are used to find the patterns, behaviour and help derive expectations of the data for further analysis and understanding [33].

One of the best examples of unsupervised learning is known as clustering, which has been found to be particularly important in performing dimensional reduction [33]. Provided the optimal quantity and quality of data to Machine Learning algorithms is important, as too little information will result in an unreliable model. Conversely, if too much of the same information is presented to a model, it is likely to become over-fit [33]. Over-fitting can be caused by a number of factors, one of which is when a model is fitted with too many features, thus resulting in an unstable model with unreliable results when the model is presented with new data. Clustering is a set of algorithms which divides a data set into a number of clusters based on their similarities given their respective feature observation values [43]. More specifically, by

performing Hierarchical Clustering on a large set of features, it is possible to group similar features together and then fit each cluster separately to a model [44]. This serves to avoid over-fitting when using too many features that are distinctly different in behaviour [33]. The goal when using Hierarchical clustering is to analyse the group of features belonging to each cluster separately with the expectation that better model performance will be a result.

2.6 Software Defects And Machine Learning

Various parties have proposed applying the use of Machine learning to the research of Static Analysis metrics and rules [6, 7]. In one research case, an analysis on topics on the source code of various software applications was performed. By categorizing the topics and performing statistical analysis, they were able to use the topics metrics on top of the existing software metrics in order to better predict defect outcomes. What was found was promising: If a particular source code file had a reputation for having introduced defects in the past, then its reputation continues into the future and thus has a high chance of contributing new defects later on in development or testing. Additionally, source code files that had a large number of topics recorded also demonstrated a high defect rate. These results suggest that Static Analysis measures may have a promising impact on the prediction of software defects [6].

A recent experiment demonstrated some significantly positive results through analysing 12 NASA datasets using deep learning Neural Network configura-

tions. The research found strong correlations between some of the static analysis metrics and predicting the outcome of defect detection. The findings also suggest that machine learning algorithms outperform traditional regression models and in most cases were statistically significant in predicting software defect outcomes [7].

The research performed on software defect prediction using Static Analysis features has generally focused on a relatively small feature set: 5 in one [6] and 21 features in another[7]. While it is generally good practice to keep a model feature set small and simple in order to avoid unnecessary complexity and potential over-fitting [36], the large number of metrics and rules that are produced by a Static Analysis tool such as SonarQube could be useful for defect prediction and avoidance, but because of that large number, are not being taken to their full potential due to the effects of Metrics Galore [33].

The goal of this thesis is to develop a statistically significant prediction model for software defects using Static Analysis metrics and rules and provide guidance on which subset of the Static Analysis metrics and rules are actually important in producing this prediction. To this end, the results will allow software development teams focus on a manageable set of metrics and rules and work toward improving their measures in order to avoid, mitigate or fix software defects as early as possible in order to reduce overall program development costs. Finally, software program managers can use the prediction models as a means to monitor and estimate the progress of a software development program more accurately than they do currently, thus providing

better insight into the health of a development program.

2.7 Measurements of Model Performance

The most common performance measurement of a regression model that is used to predict a numerical value is through the inspection of the Root Mean Squared Error (RMSE) value of the model fit [37]. This measurement is a mean measurement of the model error in predicting a particular value given a set of input features from the test set. The RMSE is an absolute error measurement, in that it is calculated by the square of the error, thus ignoring whether the error was positive or negative from the actual value. The RMSE will be the primary measure of regression model performance in this analysis. When RMSE values are similar, additional measurement values are analysed. One such measurement is R^2 (RS) (2), that is, the coefficient of determination. The R^2 value provides a percentage measurement of the coverage of output variable by the model. This is to say that the model is able to predict the R^2 percentage of variability of the problem set given the available inputs. A lower RMSE value is preferred while a larger R^2 value is preferred [37]. Another measure of regression model performance is the Mean Absolute Error (MAE) (3) which is an averaged error measurement. Likewise, Standard deviations for each of these measurements is also included: Standard Deviation of R-Squared (R.SD), Standard Deviation of RMSE (RSSD), and Standard deviation of MAE (M.SD) [37].

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n} e_i^2} \tag{1}$$

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} \tag{2}$$

$$MAE = \frac{\sum_{i=1}^{n} \mid e_i \mid}{n} \tag{3}$$

The most comprehensive performance measurement of a classification model is through the use of the Receiver Operator Characteristic Curves (ROC) [37]. The calculations leading to the creation of an ROC curve for a particular model takes into consideration several aspects of the performance of the model. These considerations are as follows:

- The Recall (Sensitivity) of the Model(4) - The number of correct predictions of an event over the total number of occurrences of the event. This measurement is also known as the true positive rate of predictions of the model [37].
- The Specificity(5) - The number of correct predictions of the absence of an event over the total number of absences of the event. This measurement is also known as the true-negative rate of the model [37].

$$Sensitivity = Recall = \frac{TruePositivePredictions}{TotalRealPositiveEvents} \tag{4}$$

$$Specificity = \frac{TrueNegativePredictions}{TotalRealNegativeEvents} \tag{5}$$

By plotting the sensitivity versus the specificity of a model, an ROC curve is created. The goal in this analysis is to maximize the performance of the model by maximizing both the sensitivity as well as the specificity. In doing so, the Area Under the Curve (AUC) must be maximized. The maximum possible value for an AUC is 1, and thus model performance will be based on the model which produces the highest AUC value [37]. Given existing literature and studies in the area of Static Analysis and Software Defects, the minimum acceptable threshold for an AUC in the Binary and Multi-Class Classification analysis in this thesis shall be 0.683 as was the minimum found in similar current research [7].

While the AUC alone is a reasonably strong indicator of model performance, two other model measurements will be discussed during the analysis and the implications of their results will be determined. These measurements are:

- The Precision(6) - The number of correct predictions of an event over the total number of predicted occurrences of the event. This measurement is also known as the positive predictive value of the model [37].
- The F-Score(7) - The harmonic mean of the Precision and Recall of the model. The F-Score is also used alongside the AUC measurement as an overall model performance indicator [37].

$$Precision = \frac{TruePositivePredictions}{TruePositivePredictions + FalsePositivePredictions} \quad (6)$$

$$F\text{-}Score = 2 * \frac{Precision * Recall}{Precision + Recall} \quad (7)$$

Finally, the AUC is used to determine variable importance in the classification model results. The Caret R model is used to calculate the variable importance, and this library accomplishes this calculation using the trapezoidal rule calculation of the AUC of the ROC curve for each feature in each model. In the case of multi-class classification, this comparison is performed in a pair-wise fashion for each feature pairs in each model. The maximum AUC value in these pair-wise comparisons is used as the variable importance for the given model. These results are then averaged out across the k-fold cross validation results [48].

3 Methodology

Analysis has been performed on a data set consisting of software metrics and rules data collected from three large-scale software development programs. A "software development program" in this context implies the execution of a contract-based software development effort with the purpose of delivering a suite of software applications to an end customer. These programs consist of defence domain software with high-reliability, high-criticality and safety-critical performance requirements, and thus were extensively tested during and after software development activities. Many individual software applications were written for each of these programs, all using the Java programming language and object-oriented design methodologies. The overall size of each software development program is captured in Millions of Source Lines of Code (MSLOCs). Refer to Table 1 for some key details of each program:

Measure	**Program 1**	**Program 2**	**Program 3**
Duration (Years)	2	1	7
Team Size (People)	30	10	80
Applications	290	132	352
LoC (MSLOCs)	4.28	2.96	7.72
Software Defects	2629	542	14926

Table 1: Important Measurements by Program

While Program 1 and Program 2 above were developed in an iterative and Agile methodology, Program 3 was developed using a Waterfall process. Despite the differences between the programs, in all cases, the software development activity was conducted as a separate phase from the final acceptance test-

ing phase of the programs. While low level software testing was conducted through the development of all three of these programs, the large majority of the software defects were discovered during the acceptance testing phase of each program.

In all programs, the number of software defects that were discovered far exceeded the original expectations of the program plans, which caused schedule and budget impacts to the development company. In an effort to avoid similar impact and surprises in the future, it is the goal of the analysis to build an effective defect prediction model as well as a strategy to focus effort on particular Static Analysis metrics and rules that could lead to better software defect rate results.

3.1 Data Set Generation Theory

The data used are snapshots of a continuum of ever-changing values. In order to limit the scope of the research in this thesis to a manageable size, it was determined that a single snapshot of software source code would be analysed and a single snapshot of software defect data would be analysed.

In order to determine the optimum point in time to perform both of these snapshots, intimate knowledge of the program plans was required. The theory built around these data collections is based on two key points:

1. The source code should be analysed after major software development was completed, but prior to acceptance testing was performed.

2. The software defect measurements should be analysed well after accep-

tance testing was performed.

The justification for point (1) above is the following:

- In order to generate a model that can be used for future programs, the metrics and rules must be based on historical data.
- Since only one snapshot was taken per program, best results are generated by choosing a snapshot that has a maximum of undiscovered software defects - this should be after major development is complete but before acceptance testing has started.

The justification for point (2) above is the following:

- The software defect values are the outcomes that are to be predicted by the model, and thus the snapshot of the defect rates was taken after most defects had been discovered, this is to ensure that the model accounts for as many defects as possible.
- Since these programs are completed, it was possible to profile their defect detection rates in order to ensure that a reasonable time snapshot was chosen.

It was important to ensure that the snapshots of the metrics and rules, and of software defects were taken at the optimal times, as it is the best way to maximize the number of latent defects in the SonarQube data, as well as minimizing the number of undiscovered defects in the defect rate report. By following the process outlined above, the features can be treated as current values while the defect rates can be treated as future values, thus structuring the model in a predictive format.

3.1.1 Source Code Snapshot Decision Analysis

Choosing a snapshot of the software source code was made possible through the use of version control. All software applications analysed were maintained in a source code version repository which consisted of a full historical record of all changes made to the repository from inception until completion. In order to perform this step, intimate knowledge of the program execution schedule was required. This detail will not be discussed in this report as it is considered proprietary to the company that provided the data.

3.1.2 Software Defect Snapshot Decision Analysis

The selection of the software defect measurements is a simpler analysis than the source code selection, as most software development programs demonstrate a similar defect detection trend [45]. By analysing the defect raise trend, and ensuring that the defect detection rates have settled to near-zero by the time of collection, it can be confirmed that all major software defects have been detected in the snapshot and should suffice for the purposes of this analysis [45].

The software defects rates for each of the three programs can be seen in the following charts. The detection rates are noted as Software Problem Reports (SPRs) and these are analogous to software defects in this context.

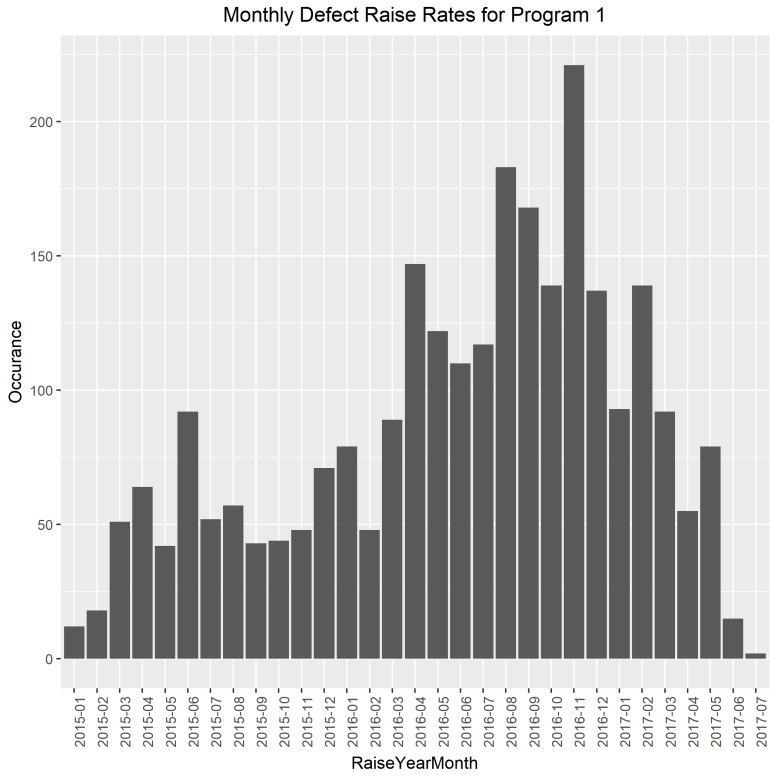

Figure 1: Monthly Software Defect Detection Rates for Program 1

For Program 1, it is apparent from Figure 1 that the defect detection rates demonstrate a well defined completion time around 2017-07. This snapshot can be used with a high degree of confidence that all major defects have been discovered and thus the data set will aid in prediction of such defects given the metrics and rules collected.

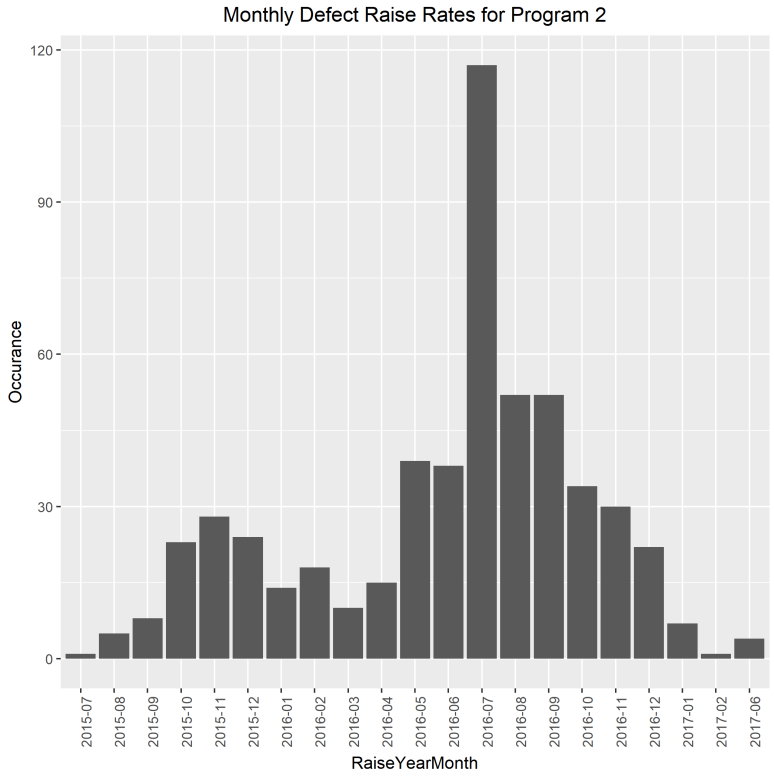

Figure 2: Monthly Software Defect Detection Rates for Program 2

Conversely for Program 2, as demonstrated in Figure 2, the defect detection rates do not demonstrate as pronounced of a completion downturn as in Program 1. Instead, the defect detection slowly ramps down until around 2017-02. At this point, the snapshot can be used with a reasonably high degree of confidence that all major defects have been detected.

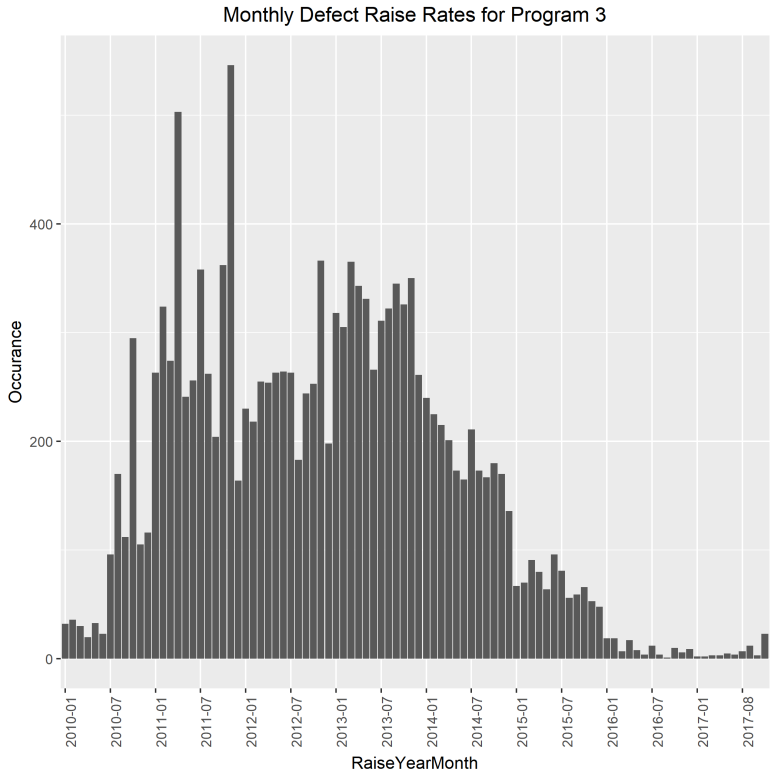

Figure 3: Monthly Software Defect Detection Rates for Program 3

Finally, for Program 3, as shown in Figure 3, there are several step downs until the defect detection rates settles out around 2016-02. Given the longer time-frame for this program, there is a very high degree of confidence that all major defects have since been detected.

The predicted outcome for each software application will be the total number of software defects detected in that corresponding application by the completion of the major software testing. This software defect count will be the target for the regression analysis in this thesis and will form the basis for the classification analysis as well.

3.2 Feature Selection Theory

Once the data for software metrics, rules, and software defects were collected, they were combined into a table of predictors and outcomes. The predictors correspond to the static analysis metrics and rules collected from a SonarQube analysis of the software source code as discussed earlier. The outcomes correspond to the number of software defects detected per software application in each repository.

Since there are several hundred static analysis metrics and rules that could be counted in the analysis, it was necessary to narrow down the predictors to a more manageable number in order to aid in analysis as well as to avoid model over-fitting [36].

3.2.1 Feature Elimination

The metrics and rules collected from the SonarQube analysis are derived from an SQL database which stores the data in an easily extractable manner. The metrics reports are stored in one table, while the rule reports are stored in another. Each rule and metric is identified with a unique identifier, a key, and another set of tables exists in the database which contain the rule and metric definitions. Additionally, SonarQube simplifies data storage such that any discretely measured software metric or rule that does not result in a single observation in the analysis will not be recorded in the data export. This eliminates a significant number of the possible metrics and rules from the analysis at the outset of the data export steps as not all rules or metrics have

resulted in observations. Refer to Table 2 for the first five features that were removed from the analysis due to lack of detection.

Feature ID	**Description**
X136	Security - Servlet reflected cross site scripting vulnerability
X137	Bad practice - Method may fail to close database resource
X138	Sequence of calls to concurrent abstraction may not be atomic
X139	Bad practice - Comparison of String parameter using == or !=
X140	Bad practice - Transient field that isn't set by deserialization.

Table 2: Detail of a Sample of Five Features Not Detected in SonarQube Analysis

Next, once the data is imported into R for analysis, it is possible to further eliminate insignificantly contributing features by analysing the variance of the values for each feature. These predictors are called zero-variance and near-zero variance predictors. Zero-variance predictors add no value to a model as they are essentially a constant, keeping them would just add complexity to a model and potentially contribute to a weaker fit [37]. Near-zero variance predictors have the potential of becoming zero-variance predictors during cross validation, which when run through most models will cause the model fit to fail due to missing data. Additionally, near-zero variance predictors are assumed to offer little value to the model as their values are mostly the same and can be safely ignored [37]. Section 4.2.1 provides further detail on this subject, including the thresholds used for this analysis.

Finally, predictor pairs which are highly linearly correlated with each other,

also called collinearity, are likewise a good target for removal. Only one feature of a pair of highly correlated features will be removed as this accomplishes two goals:

1. Eliminate unnecessary complexity from the model [37].
2. Reduce the impact on the model of the two features to one feature, avoiding potential over-fitting [37].

By taking these steps above, the goal is to eliminate any features that will either have no effect, a trivial effect, or a detrimental effect on the behaviour of the model.

3.2.2 Feature Clustering

Performing a clustering analysis on the behaviour of each feature across all values can provide insight into a means to group similar features for further analysis. By transposing the data and performing a k-means clustering analysis [46], it is possible to separate features and run each cluster separately through analysis.

Several considerations have been made in order to ensure that the results remain consistent:

1. A plot of the variability within clusters (total within-cluster sum of squares) for each cluster has been generated in order to allow for an appropriate selection of the number of clusters through the elbow method [47].

2. A minimum cluster size has been set through trial and error, any clusters that contain a feature membership less than or equal to this threshold will be joined with each other. This will allow for a more meaningful analysis as studies have shown that fitting very small numbers of features to a model can produce less favorable results [33].

By taking the steps above, the goal is to identify groups of features that behave similarly and perform further analysis on each of these clusters. By doing this, it is expected that a more accurate and useful model will be generated for each of the groups than would otherwise be created on the entire feature space as a whole.

3.2.3 Recursive Feature Elimination

In some of the clusters, the number of features still remains high and could still result in an over-fitted model if all are used when fitting. To avoid this issue, Recursive Feature Elimination (RFE) is performed using the caret library in R [48]. By recursively attempting model fits with every permutation and combination of the set of features in a cluster within a range of minimum feature set sizes, it is possible to measure the performance of each of these fits [49].

The method of RFE chosen in this study was to minimize the RMSE value through each iteration of feature sets. This is accomplished by attempting fits for each model using each set of features for each feature set length chosen. For example, if there are 10 features available, RFE will be performed

on 1 to 10 features with each feature size iteration attempt itself consisting of a set of iterations of each permutation of features. In this example, the number of features will start at 1, and in this loop, every feature is fit to the model separately, with the results of each fit recorded. Next, RFE attempts 2 features, this time iterating over every feature combination pair. This continues until reaching 10 features, where all features are fit at the same time. After the entire execution is completed, the performance of each fit is ranked by RMSE and the best feature set size and feature set is chosen [48].

By performing the recursive feature model fitting, it is possible to compare the adjusted R-squared value, for regression, or the accuracy value for classification, of the resultant model for each run. In doing so, the results will suggest the following:

1. The optimal number of features to use for fitting to the model
2. The list of the best features to use in fitting the model

By performing the process above, the goal is to arrive at an optimal feature-set that will provide the most valuable information about predicting the outcome of software defects while at the same time offering the least-complex solution to the problem, thus avoiding over-fitting [37].

3.3 Prediction Analysis Theory

The predictive analysis performed can be summarized across the following categories:

1. Regression
2. Binary Classification
3. Multi-class Classification

3.3.1 Regression Modelling

Regression is performed on the aggregated defect value directly and is used to predict the actual number of defects per unique application using the metrics and rules as features. This analysis is performed using the following models:

1. Linear Regression Model
2. Neural Network
3. Linear Support Vector Machine

It should be noted that while the only results from Support Vector Machines are with a Linear Kernel, other kernel types were attempted during this research. It was determined that radial and polynomial SVM Kernels did not perform well and thus, the results of those analysis are omitted from this thesis.

3.3.2 Binary Classification Modelling

Binary classification is performed on a binary variant of the software defect value. By converting the software defect count to a binary definition by answering the following question: "Does the application contain a defect?".

If the answer is yes, then the value is set to true, if not, the value is set to false. This allows for classification to be performed, and thus the following models are explored:

1. Decision Tree
2. Random Forest
3. Neural Network
4. Linear Support Vector Machine

Class imbalance must be taken into consideration when performing fitting with the above models in a binary classification scheme [37]. As shown in Table 3, the number of software applications with at least one defect far outnumber those that do not have a single defect. Given this class imbalance, classifiers can sometimes favour the outcome that is most likely, and in extreme cases, always predict that outcome, regardless of the inputs [37].

Classification	**Program 1**	**Program 2**	**Program 3**
HAS_SPR	73.10%	60.15%	81.84%
NO_SPR	26.90%	39.85%	18.16%

Table 3: Class Imbalance of Software Defect Occurrence in Software Applications by Program

In order to avoid class imbalance side-effects, up-sampling of the minority class data points will be performed in order to compensate for the class imbalance issues [50]. By randomly adding additional data points that are repeats of the minority class until both classes are equivalent, it will force the models to perform more insightful fitting. Up-sampling will be accomplished

through the use of the preProcess method in the Caret library in R [48]. Caret accomplishes up-sampling by randomly sampling the minority class with replacement, until the number of samples of the minority class are approximately equivalent to the majority class. Up-sampling alone has the potential of introducing inaccuracies in the model, and thus all analysis will be done with repeated cross validation in order to ensure a consistent model is produced.

3.3.3 Multi-class Classification Modelling

The selection of the classes was done with the overall software defect distribution in mind. Refer to Figure 4 for a scatter plot of the defect counts for each software application.

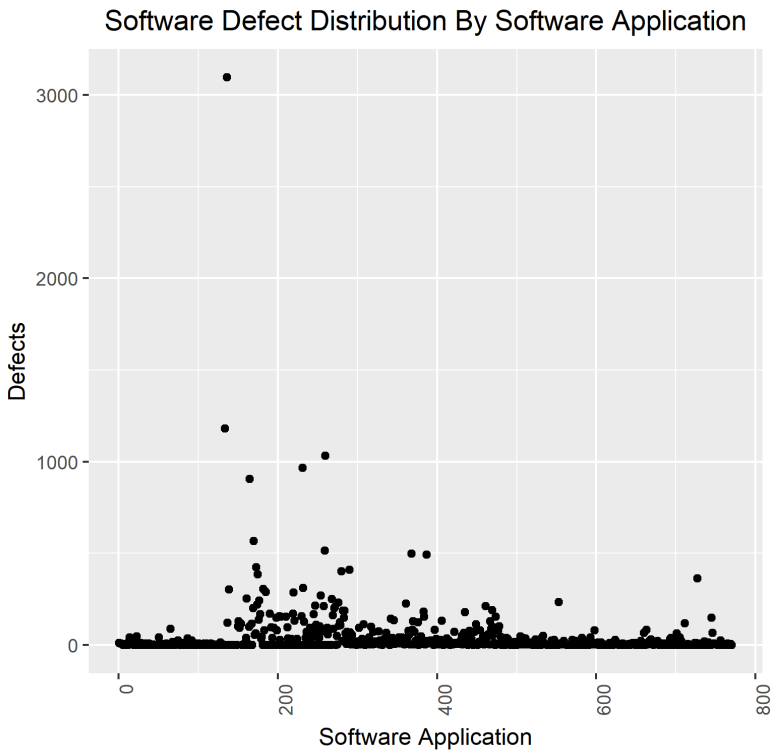

Figure 4: Scatter Plot of Software Defects by Software Application

As Figure 4 demonstrates, a large number of the software applications exhibit a low number of software defects. This behaviour makes the scatter plot difficult to interpret on its own, thus this necessitates further analysis via a density plot, see Figure 5.

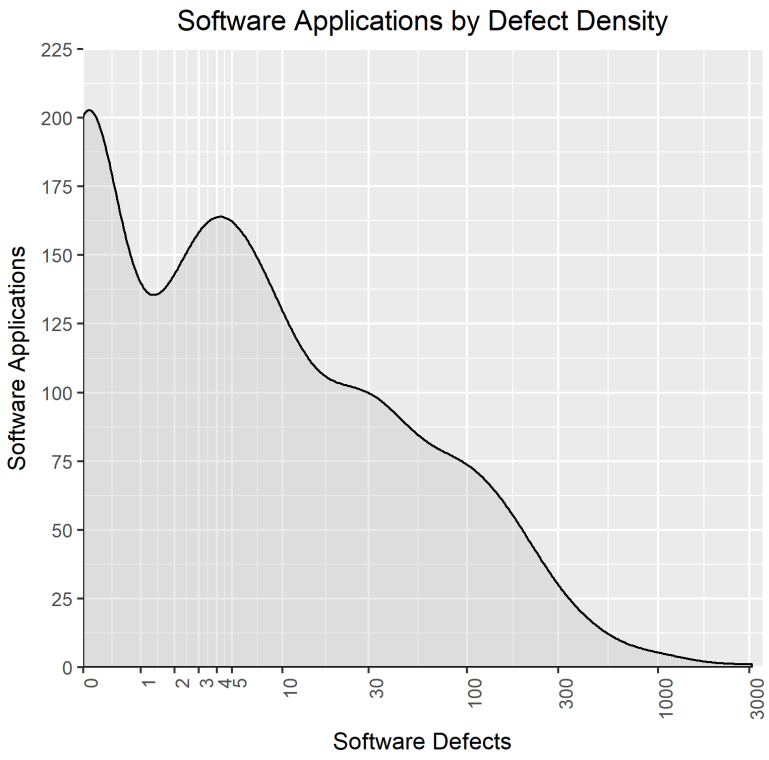

Figure 5: Density Plot of Software Applications by Number of Software Defects

Given the above distribution, the software defect rate was stratified across several value categories. These values will be defined as follows:

1. No defects
2. Low defect rate
3. Moderate defect rate
4. High defect rate
5. Extreme defect rate

The thresholds of the defect rate classes listed above will be determined based

on a statistical analysis of the distribution of the software defect rates across all applications in all three programs.

Multi-class classification is more restrictive than binary classification, as most multi-class classification models are built out of multiple binary classification models [37]. Thus, only the following models will be tested:

1. Decision Tree
2. Random Forest
3. Neural Network
4. Linear Support Vector Machine

3.3.4 Model Tuning Parameter Search

In some cases, such as with Decision Trees, Random Forests, SVMs, and Neural Networks, models require the configuration of model tuning parameters. In this thesis, only a narrow tuning parameter range will be shown in the results in order to provide a concise results summary set. The ranges of tuning parameters will be chosen using a parameter grid search technique and trial and error. Once a range of tuning parameters are found to provide best model performance, the range will be reduced to provide the results shown in the following sections.

3.3.5 Modelling Goals

By employing several different methods of analysis, the goal is to either:

- Obtain the optimal model configuration for software defect prediction or,
- Reasonably exhaust the options for exploring the solution space if no statistically significant model is found

In sum, by following the methodologies outlined above, it is anticipated that the analysis will be conducted with a sound process and will produce reliable and potentially valuable results.

4 Results

4.1 Data Preparation

4.1.1 Initial Data Set Creation

A general overview of the data collection and preparation steps can be observed in Figure 6:

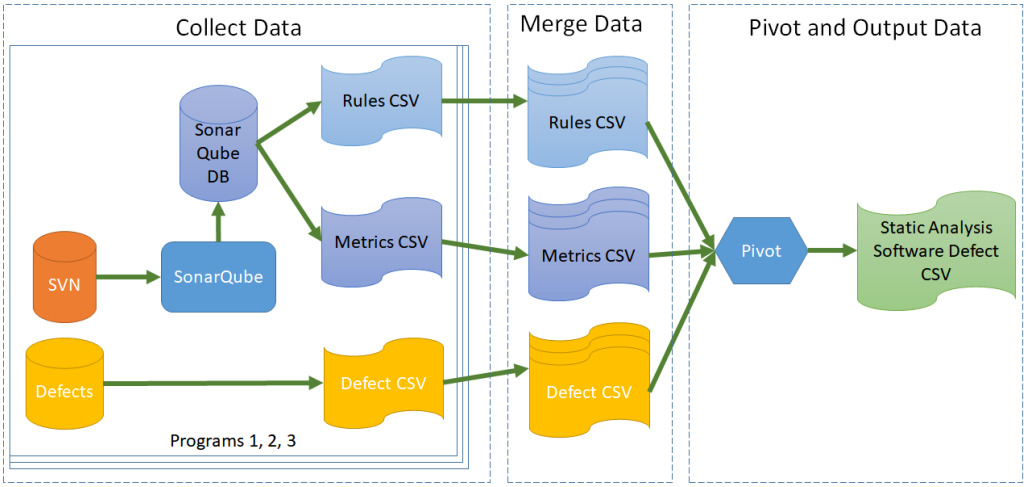

Figure 6: Data Collection and Preparation Methodology Diagram

The initial data sets were created in several steps:

1. SonarQube analysis on three separate code repositories
2. Software defect analysis on three separate software programs

Once the SonarQube analysis was performed on each code repository, the rules and metrics tables were extracted from the SonarQube SQL database. The data from each of these tables was exported to csv files for analysis.

Likewise to the SonarQube data, the software defect data was extracted from each of the software program defect databases and exported to corresponding csv files.

Once all data was extracted, it had to be joined and aggregated, as discussed below.

4.1.2 Data Joining

Each metrics and rules sets were generated using the same baseline of SonarQube, thus ensuring that the rule and metric identifiers would be consistent across each execution. This allows for a simplistic joining of the data sets from each of the three software programs via row and column binding.

The metrics tables from each of the three programs were read into R and row-bound with each other, thus producing a single metrics dataset. The same process was repeated for the rules and the software defects data sets, again, producing single tables for each of the three types of measurements being analysed in this research.

At this point, the metrics and rules tables were arranged in a tall format, which is not conducive to prediction analysis. This is to say that the tables each contained three columns:

1. The software application unique identifier
2. The rule or metric unique identifier
3. The rule or metric value

Table 4 depicts the characteristics of these tables for each program:

Program	**Metrics Rows**	**Rules Rows**
Program 1	9258	7905
Program 2	4306	3975
Program 3	10853	9421

Table 4: Raw Metrics and Rules Table Row and Column Count by Program

In order to perform effective prediction analysis, the data had to be pivoted by the rule and metric identifiers, thus producing a number of columns equal to the number of unique rules and metrics, with the value of each of these columns equal to the value of the corresponding rule or metric in the original table.

The metrics and rules data tables were pivoted vertically via the cast library in R. This then resulted in two wide data tables. With the indexes set to the software application unique identifiers, the two tables could be joined. This resulted in a new, wide, data table with a column for every rule and metric value and a row for every unique software application.

Table 5 depicts the characteristics of the pivoted metrics tables:

Table	**Observations**	**Count**
Program 1 Metrics	290	36
Program 2 Metrics	133	35
Program 3 Metrics	347	35

Table 5: Pivoted Metrics Table Observations and Counts Data

Similarly, Table 6 depicts the characteristics of the pivoted rules tables:

Table	Observations	Count
Program 1 Rules	257	129
Program 2 Rules	125	126
Program 3 Rules	312	133

Table 6: Pivoted Rule Table Observations and Counts Data

In both Table 5 and Table 6, the Observations value corresponds to the number of software applications profiled in the analysis while Count indicates the number of Metrics or Rules values that were recorded for each application.

The software defect table simply contained a list of every software defect recorded to date on each program. What is required instead, is a count of defects per unique software application. This necessitated a pivot by software unique identifier, with the aggregation method of count. At this point, the resultant table contained two columns, one of the unique software identifiers and the other of the count of software defects for the corresponding software application.

Once pivoted and aggregated, it was then possible to join the metrics and rules table with the software defect count table. The tables were aligned by the unique software identifier and joined by column. The resultant data set is a table that contains the unique software identifier per software application, a column for every metric and rule analysed and finally, a column that represents the number of defects detected for each software application. This table contains all of the necessary data and in a format that is conducive to analysis. Table 7 depicts the characteristics of the table used in the following analysis. In this table, Observations indicates the total number of software

applications profiled in the analysis while Measures indicates the total number of Metrics and Rules measurements collected in the analysis:

Stat	**Count**
Observations	770
Measures	170

Table 7: Static Analysis Software Defect Table Count by Observations and Measurements

4.2 Feature Selection

The feature space was optimized using the following methodology:

1. Remove features that will either have no impact or a negative impact on modelling
2. Group features using k-means clustering
3. Eliminate features using recursive feature elimination

Refer to Figure 7 for a depiction of the steps taken in the feature selection process.

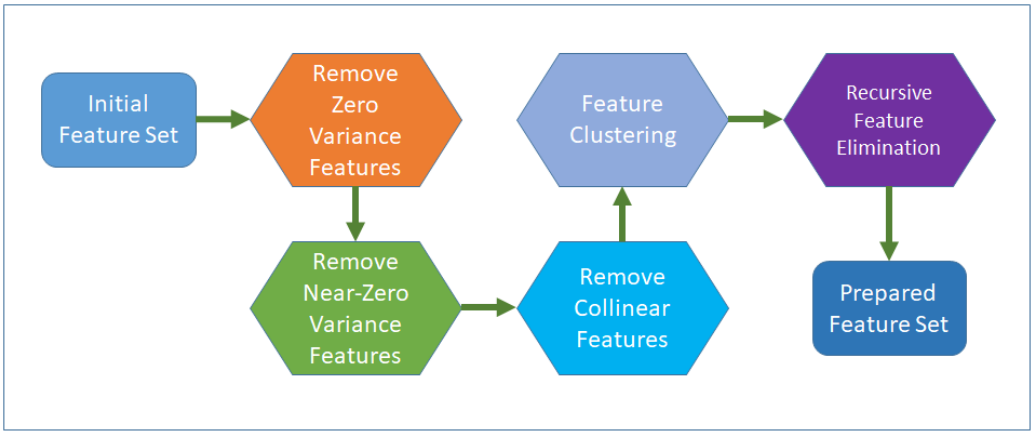

Figure 7: Feature Selection Methodology Diagram

Refer to Table 56 in Appendix B for the full list of rules and metrics used in this analysis.

4.2.1 Feature Removal Results

The first analysis performed was with respect to the variance of the predictors in the Static Analysis Software Defect table which was described in Table 7. This was broken into two parts: zero variance and near-zero variance predictors.

Eliminating the zero variance predictors was accomplished by selecting the columns from the main data set with the number of unique values set to one. Refer to Table 8 for the features that were removed using this process.

Feature ID	Description
X4	Lines of code per language
X27	Functions distribution /complexity
X28	Files distribution /complexity
X118	Directory cycles
X119	Directory tangle index
X120	File dependencies to cut
X121	Package dependencies to cut
X122	Directory edges weight
X132	Technical Debt on new code
X134	SQALE Development Cost

Table 8: Details of Features Removed Due to Zero Variance

Next, eliminating near-zero variance features required the definition of "near-zero" by means of a minimum variance threshold. The thresholds chosen in this analysis were done so using typical values as per the literature [37]. These thresholds were as follows:

1. Ratio of the most common feature over the second most common feature must be less than: $\frac{95}{5}$
2. The number of unique values in a feature is less than: 10

Using the parameters described above, the features that were removed are shown in Table 9 below.

Feature ID	Feature ID	Feature ID
X626	X704	X766
X631	X705	X767
X632	X708	X770
X637	X713	X772
X640	X714	X775
X645	X715	X777
X647	X720	X779
X648	X723	X792
X651	X725	X795
X653	X726	X801
X655	X731	X814
X660	X732	X817
X667	X739	X820
X671	X740	X821
X675	X742	X822
X677	X744	X824
X679	X746	X825
X681	X747	X828
X682	X748	X830
X687	X754	X832
X689	X756	X836
X690	X757	X838
X693	X762	X839
X703	X764	X845

Table 9: IDs of Features Removed Due to Near Zero Variance

At this point, it is possible to analyse the linear correlation coefficients of each feature and the outcome. This analysis in its entirety can be seen in Figure 18 in Appendix A. Note that features that are not significantly correlated with each other are indicated by a large black 'X'. Features with a significant positive correlation are indicated with a blue circle. Features with a significant negative correlation are indicated with a red circle. In either positive or negative cases, the stronger correlations are indicated with larger

circles. Table 10 lists the top 10 features that are most strongly correlated with the outcome. It is anticipated that these features will prove important in the models fitted later in this thesis.

Feature ID	**Description**	**C.Coef.**
X641	Declarations should use Java collection interfaces such as List rather than specific implementation classes such as LinkedList	0.2605
X841	TODO tags should be handled	0.2603
X727	Exception handlers should preserve the original exception	0.2391
X734	Methods should not be too complex	0.2381
X11	Accessors	0.2346
X823	Tabulation characters should not be used	0.2331
X8	Directories	0.2288
X788	Variables should not be declared and then immediately returned or thrown	0.2268
X7	Files	0.2213
X683	Avoid commented-out lines of code	0.2110

Table 10: Top Ten Features Correlated with Outcome

Removing the highly inter-correlated features required a correlation analysis to be performed on the entire feature space using the Pearson Correlation Coefficient. This correlation analysis provided insight into the inter-correlations between features in a linear space. Doing this results in a complex correlation plot that can be seen in Figure 18 in Appendix A. Additionally, Table 11 lists the top 10 most highly inter-correlated features in the data set. This table is filtered on magnitude of the correlation coefficient only, regardless as to whether the correlation is positive or negative. This view provides a filtered and focused snapshot of the most inter-correlated features.

Feat. ID A	**Description A**	**Feat. ID B**	**Description B**	**C.Coef.**
X423	Duplicated blocks	X89	Duplicated files	1
X24	Complexity in functions	X20	Complexity	1.0000
X13	Public API	X10	Functions	0.9984
X20	Complexity	X3	Lines of code	0.9963
X24	Complexity in functions	X3	Lines of code	0.9962
X3	Lines of code	X1	Lines	0.9948
X12	Statements	X3	Lines of code	0.9924
X20	Complexity	X1	Lines	0.9923
X24	Complexity in functions	X1	Lines	0.9923
X20	Complexity	X12	Statements	0.9895

Table 11: Top Ten Inter-correlated Features

In order to remove features with high inter-correlation, a linear correlation threshold had to be selected. For the purposes of this analysis, a threshold of 75% was chosen both as it is a typical threshold in similar analyses [37] as well as it achieved the best results from other thresholds attempted in this research. From this correlation matrix, Table 12 depicts the features removed.

Feature ID	Feature ID	Feature ID
X3	X131	X774
X6	X135	X781
X7	X423	X782
X8	X634	X785
X10	X649	X787
X11	X673	X788
X12	X686	X790
X13	X692	X796
X15	X697	X804
X18	X710	X809
X20	X727	X827
X21	X734	X831
X24	X738	X834
X87	X741	X842
X88	X753	
X89	X755	

Table 12: IDs of Features Removed Due to High Intercorrelation

After all feature removal techniques are completed, there are 39 features remaining for further analysis. This will start with Feature Clustering, Recursive Feature Elimination and then Regression and Classification model fitting as shown in the following sections.

4.2.2 Feature Clustering Results

In order to cluster the features, a number of clusters must be chosen. To do this, a range of cluster values, k, was run through a k-means analysis and the total sum of squares value (8) for each k-value was captured. Figure 8 depicts the relationship between total sum of squares per k-value chosen.

$$TSS = \sum_{i=1}^{n} (y_i - \bar{y})^2 \tag{8}$$

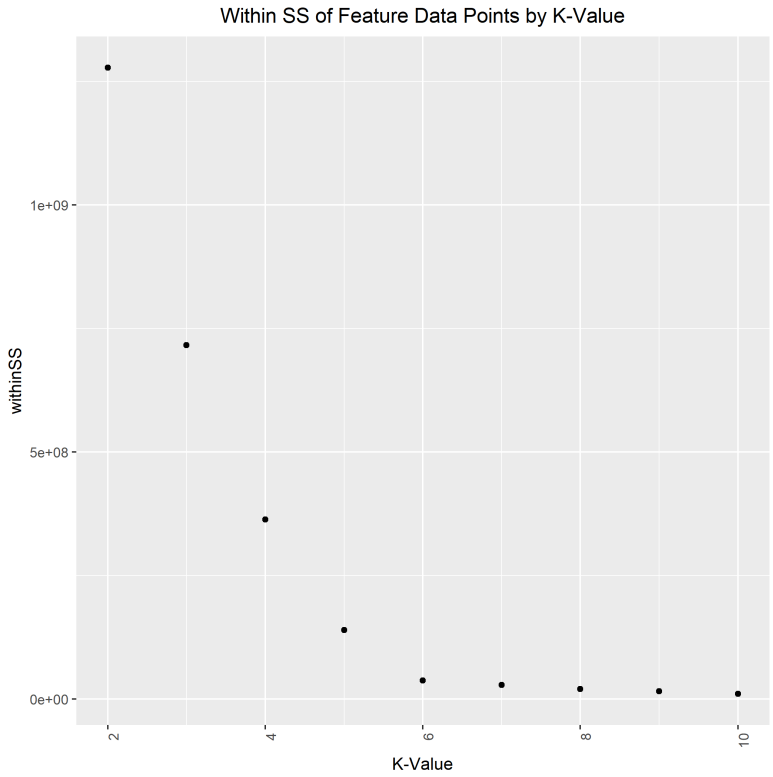

Figure 8: Within Sum of Squares Totals by K-Value

Choosing the number of clusters at one k-value higher than the elbow of the curve, 7 clusters are chosen [47]. The k-means analysis was performed again for 7 clusters and each feature was assigned membership to a particular cluster k-value. The distribution of the features by k-value can be seen in Figure 9.

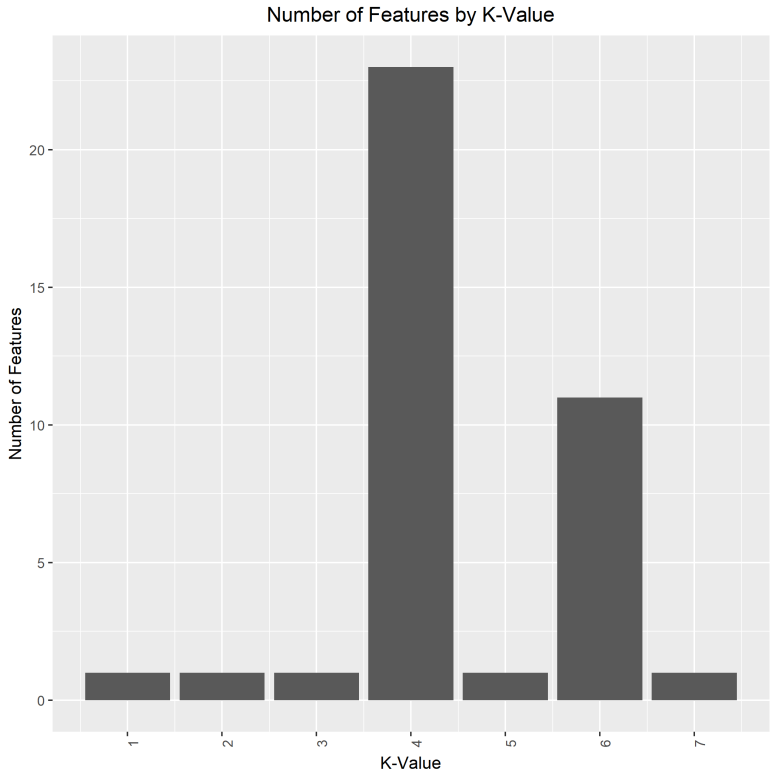

Figure 9: Feature Count by Cluster Membership

It can be seen that two clusters contain a reasonable quantity of features while the remaining five clusters only contain a single member each. The minimum threshold of membership per feature group was chosen at 5, and thus, five of the clusters were merged together. By performing this cluster merging, Figure 10 depicts the final cluster membership configuration:

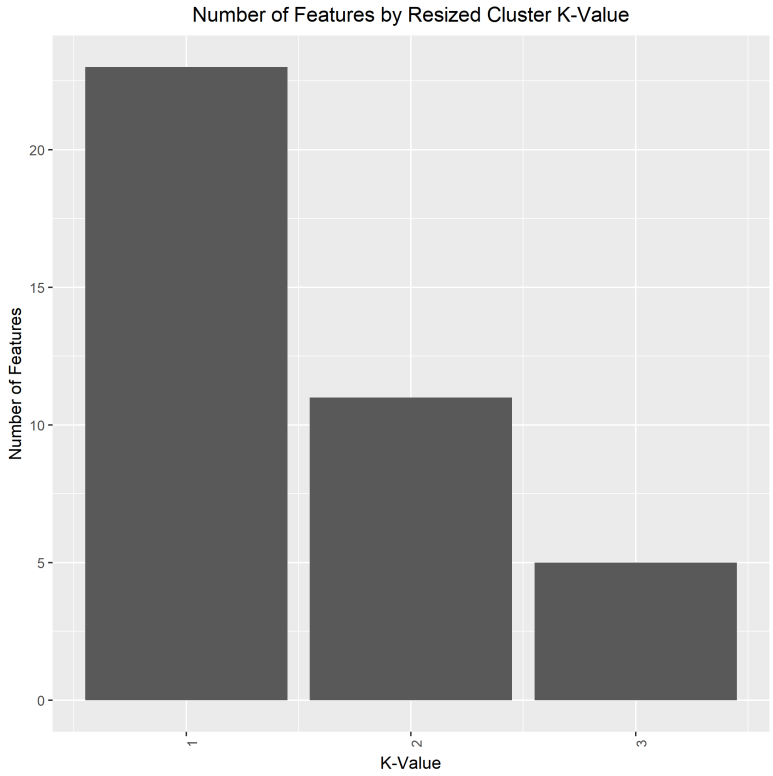

Figure 10: Revised Feature Count by Cluster Membership After Applying Minimum Cluster Size

Table 13 lists the feature membership for Cluster 1:

Feature ID	Description
X133	SQALE Rating
X16	Comments (%)
X17	Public documented API (%)
X23	Complexity /class
X25	Complexity /function
X628	equals(Object obj) and hashCode() should be overridden in pairs
X644	Loggers should be private static final and should share a naming convention
X661	Generic wildcard types should not be used in return parameters
X700	Math operands should be cast before assignment
X709	Loops should not contain more than a single break or continue statement
X729	Empty arrays and collections should be returned instead of null
X751	Local variables should not shadow class fields
X760	Expressions should not be too complex
X761	Lambdas and anonymous classes should not have too many lines
X763	Unused private fields should be removed
X769	Constants should be declared final static rather than merely final
X771	Unused method parameters should be removed
X789	Return of boolean expressions should not be wrapped into an if-then-else statement
X805	A field should not duplicate the name of its containing class
X810	Try-catch blocks should not be nested
X815	Nested blocks of code should not be left empty
X837	Deprecated code should be removed eventually
X90	Duplicated lines (%)

Table 13: Detailed List of Features in Cluster 1

Similarly, Table 14 lists the feature membership for Cluster 2:

Feature ID	Description
X641	Declarations should use Java collection interfaces such as List rather than specific implementation classes such as LinkedList
X680	String literals should not be duplicated
X683	Avoid commented-out lines of code
X699	System.out and System.err should not be used as loggers
X736	Switch cases should not have too many lines
X758	Collapsible if statements should be merged
X808	Fields in a Serializable class should either be transient or serializable
X819	Right curly braces should be located at the beginning of lines of code
X835	Strings literals should be placed on the left side when checking for equality
X841	TODO tags should be handled

Table 14: Detailed List of Features in Cluster 2

Finally, Table 15 lists the feature membership for Cluster 3:

Feature ID	Description
X665	The members of an interface declaration or class should appear in a pre-defined order
X733	Method names should comply with a naming convention
X793	Useless imports should be removed
X823	Tabulation characters should not be used
X844	Constant names should comply with a naming convention

Table 15: Detailed List of Features in Cluster 3

As shown in the above tables, there is no obvious pattern in the feature groupings in each Cluster. The following sections analyse each cluster in further detail.

4.2.3 Recursive Feature Elimination Results

For each regression model, each permutation of all features in the cluster is analysed for best fit. The results in the following sections depict the number of features used in the fit as "No.Feat.".

Linear Regression

The results of the RFE performed on the linear regression models generated for each set of features in each cluster are depicted in the following tables.

No.Feat.	RMSE	RS	MAE	R.SD	RSSD	M.SD
1	120.0182	0.0515	52.6530	87.0148	0.0848	12.9153
2	**119.4884**	**0.0876**	**52.2050**	**87.7059**	**0.1389**	**13.5808**
3	121.0157	0.0666	52.7023	86.5923	0.0898	13.2556
4	121.5295	0.0711	52.8622	85.9967	0.0873	13.2140
5	122.9176	0.0683	53.1457	85.4873	0.0787	13.2430
6	125.6524	0.0592	53.8689	85.2324	0.0694	13.2741
7	129.3948	0.0594	54.8496	86.0261	0.0716	13.7886
8	131.8273	0.0539	55.5778	85.8978	0.0645	14.0532
9	132.6767	0.0507	56.0697	85.4009	0.0652	13.9811
10	133.8371	0.0458	56.6911	85.3318	0.0620	14.2631
11	134.7290	0.0415	57.2266	84.9935	0.0582	14.3732
12	134.9811	0.0409	57.3252	84.8086	0.0582	14.2936
13	134.6672	0.0443	57.1374	84.4388	0.0637	14.2511
14	134.0804	0.0444	56.9282	84.5188	0.0646	14.2596
15	134.1463	0.0455	56.9825	84.5355	0.0654	14.3233
16	133.6273	0.0464	56.8974	84.4201	0.0662	14.3418
17	133.5611	0.0468	56.9622	84.3501	0.0679	14.3081
18	133.3260	0.0462	57.0377	84.2773	0.0665	14.2898
19	133.3637	0.0460	57.2118	84.2533	0.0663	14.2778
20	133.2631	0.0462	57.2497	84.2859	0.0667	14.2315
21	133.1022	0.0463	57.2170	84.3823	0.0673	14.2603
22	133.0347	0.0468	57.3082	84.3722	0.0670	14.3105
23	132.8985	0.0474	57.2531	84.4241	0.0675	14.3508

Table 16: Linear Regression RFE Performance Results - Cluster 1 Features

As shown in Table 16, the best performing configuration for Cluster 1, with the lowest RMSE value is with two features.

No.Feat.	RMSE	RS	MAE	R.SD	RSSD	M.SD
1	**117.9568**	**0.1243**	**51.2574**	**86.4188**	**0.1667**	**13.1916**
2	118.5452	0.1402	51.6701	86.1029	0.1825	13.5505
3	120.3171	0.1257	52.4586	85.6742	0.1631	13.6824
4	121.4376	0.1282	52.7184	84.8881	0.1655	13.5102
5	121.0671	0.1299	52.6847	84.9532	0.1683	13.5406
6	120.7130	0.1285	52.7024	85.1857	0.1699	13.5910
7	120.4337	0.1282	52.6233	85.2804	0.1670	13.5552
8	120.6840	0.1278	52.7525	85.1106	0.1662	13.5026
9	120.9211	0.1281	52.8636	85.0113	0.1683	13.5005
10	120.8802	0.1292	52.8795	84.9995	0.1687	13.5122
11	120.8006	0.1307	52.8545	84.9820	0.1703	13.4980

Table 17: Linear Regression RFE Performance Results - Cluster 2 Features

Similarly, Table 17 indicates that the best performing configuration for Cluster 2 is with a single feature as this result has the lowest RMSE value.

No.Feat.	RMSE	RS	MAE	R.SD	RSSD	M.SD
1	119.9199	0.0635	53.3861	87.1557	0.0957	13.0542
2	121.1856	0.0274	53.5315	86.8848	0.0481	13.2898
3	120.3196	0.0517	52.8067	87.1525	0.0894	13.4784
4	**119.8153**	**0.0719**	**52.7282**	**86.8557**	**0.1184**	**13.4334**
5	119.9476	0.0696	52.7745	86.8727	0.1162	13.4864

Table 18: Linear Regression RFE Performance Results - Cluster 3 Features

Conversely, Table 18 shows that the best performing configuration for Cluster 3 is with four features.

Given the results above, the RFE analysis on the linear regression model recommends the seven features shown in Table 19.

Feature ID	Cluster	Description
X133	1	SQALE Rating
X709	1	Loops should not contain more than a single break or continue statement
X841	2	TODO tags should be handled
X665	3	The members of an interface declaration or class should appear in a pre-defined order
X733	3	Method names should comply with a naming convention
X793	3	Useless imports should be removed
X823	3	Tabulation characters should not be used

Table 19: Linear Regression RFE Feature Details Based on Best Performing Fits

Neural Network

Since this is a regression analysis, a linear activation function was used in the output layer for this Neural Network. For each cluster, the Neural Network performance was measured in this Feed Forward, Single Hidden Layer Neural Network model. The tuning parameters used were as follows:

- Size (S) - The number of hidden nodes in the Neural network: varying from 8 to 10 in steps of 1,
- Decay (D) - The rate of the weight decay: varying from 252 to 254 in steps of 1. Note: Decay refers to the rate of decay of the neuron weight penalty used in the calculation of the Neural Network cost function. This allows a variation in the trade off between error cost and large weight size.

The results of the RFE performed on the Neural Network models are depicted in the following tables.

No.Feat.	RMSE	RS	MAE	R.SD	RSSD	M.SD
1	116.3476	0.0425	51.5849	91.6029	0.0430	15.1324
2	115.6268	0.0626	50.5348	91.8366	0.0524	15.1701
3	115.5237	0.0650	49.9031	91.8711	0.0536	15.1185
4	115.3905	0.0662	49.6386	91.7614	0.0508	15.0969
5	115.3888	0.0657	49.6468	91.5877	0.0495	15.0116
6	**115.2162**	**0.0676**	**49.3733**	**91.3957**	**0.0487**	**15.1552**
7	115.3363	0.0655	49.3542	91.2983	0.0476	15.1084
8	115.5223	0.0617	49.3556	91.2685	0.0464	15.2249
9	115.5592	0.0621	49.4051	91.1674	0.0481	15.1441
10	115.4936	0.0648	49.3441	91.0498	0.0526	15.1503
11	115.6213	0.0623	49.3371	91.0393	0.0517	15.2426
12	115.7327	0.0619	49.4554	90.9665	0.0516	15.2139
13	115.6221	0.0628	49.3558	90.8758	0.0514	15.2309
14	115.5088	0.0649	49.3080	90.9438	0.0547	15.2995
15	115.5674	0.0657	49.2885	90.8184	0.0566	15.0780
16	115.6821	0.0626	49.2163	90.9242	0.0544	15.2619
17	115.5765	0.0648	49.1363	90.7512	0.0570	15.1295
18	115.7893	0.0608	49.3834	90.8769	0.0522	15.2996
19	115.9967	0.0592	49.5271	90.7997	0.0540	15.2642
20	115.8646	0.0612	49.2960	90.8361	0.0551	15.3611
21	116.0249	0.0585	49.4587	90.6913	0.0491	15.1451
22	116.2057	0.0554	49.4026	90.7028	0.0505	15.2322
23	116.0526	0.0583	49.2633	90.6590	0.0523	15.1875

Table 20: Neural Network Regression RFE Performance Results - Cluster 1 Features

As shown in Table 20, the Neural Network model had the best performance with six features for Cluster 1.

No.Feat.	RMSE	RS	MAE	R.SD	RSSD	M.SD
1	117.1593	0.0931	49.6487	88.4655	0.0588	13.3960
2	116.6578	0.1070	49.2592	88.5293	0.0695	13.3188
3	**116.6002**	**0.1072**	**49.1913**	**88.5392**	**0.0699**	**13.2193**
4	116.6032	0.1073	49.2135	88.4798	0.0683	13.1069
5	116.6048	0.1081	49.2936	88.4614	0.0694	13.0742
6	116.6972	0.1059	49.3741	88.4141	0.0684	12.9359
7	116.7925	0.1042	49.4305	88.4042	0.0668	12.9288
8	116.7759	0.1038	49.6133	88.3939	0.0676	12.9983
9	116.7816	0.1030	49.6538	88.3999	0.0668	12.9340
10	116.9656	0.0998	49.7036	88.3200	0.0650	12.9989
11	116.8309	0.1023	49.5974	88.3186	0.0666	12.9954

Table 21: Neural Network Regression RFE Performance Results - Cluster 2 Features

As depicted in Table 21, the Neural Network performed best while using three features for Cluster 2.

No.Feat.	RMSE	RS	MAE	R.SD	RSSD	M.SD
1	**118.9183**	**0.0468**	**51.7974**	**88.2942**	**0.0474**	**13.2326**
2	119.1451	0.0286	51.6721	88.2358	0.0331	13.2598
3	119.1950	0.0262	51.7126	88.1364	0.0262	13.1726
4	119.2186	0.0255	51.8040	88.1169	0.0218	13.1434
5	119.1878	0.0255	51.7683	88.0985	0.0232	13.2401

Table 22: Neural Network Regression RFE Performance Results - Cluster 3 Features

Finally, Table 22 indicates that the Neural Network performed best with a single feature on Cluster 3.

Given the results above, the RFE analysis on the Neural Network model recommends the ten features shown in Table 23.

Feature ID	Cluster	Description
X133	1	SQALE Rating
X23	1	Complexity /class
X25	1	Complexity /function
X644	1	Loggers should be private static final and should share a naming convention
X760	1	Expressions should not be too complex
X771	1	Unused method parameters should be removed
X699	2	System.out and System.err should not be used as loggers
X736	2	Switch cases should not have too many lines
X758	2	Collapsible if statements should be merged
X665	3	The members of an interface declaration or class should appear in a pre-defined order

Table 23: Neural Network Regression RFE Feature Details Based on Best Performing Fits

Support Vector Machines

The Support Vector Machine used in this analysis was a Linear Kernel with the tuning regularization parameter, 'C', tuned from 0.0012 to 0.0013 in steps of 0.000001. The results of the RFE performed on the Linear Kernel SVM models are depicted in the following tables.

No.Feat.	RMSE	RS	MAE	R.SD	RSSD	M.SD
1	121.6274	0.0245	41.2500	88.5668	0.0783	13.6673
2	121.2532	0.0254	41.2530	88.6632	0.0452	13.5831
3	120.7461	0.0432	41.1576	88.7705	0.0732	13.5713
4	120.2960	0.0599	41.0596	88.8663	0.0849	13.5498
5	119.9476	0.0802	41.0704	88.9328	0.1029	13.5581
6	119.6306	0.0977	41.0362	88.9428	0.1212	13.5256
7	119.4877	0.0965	41.0001	88.9719	0.1146	13.5749
8	119.3817	0.0988	40.9513	89.0234	0.1095	13.5526
9	119.3268	0.0988	40.9224	89.0261	0.1094	13.5325
10	119.2883	0.1015	40.8856	89.0466	0.1105	13.5800
11	119.2836	0.1054	40.8460	89.0195	0.1096	13.5671
12	119.2835	0.1055	40.8106	88.9924	0.1082	13.5909
13	119.2532	0.1079	40.8032	88.9775	0.1117	13.5937
14	119.2621	0.1085	40.7864	88.9733	0.1132	13.5954
15	119.2519	0.1123	40.7807	88.9719	0.1216	13.5863
16	119.3178	0.1076	40.7751	88.9562	0.1225	13.5755
17	119.3022	0.1072	40.7651	88.9681	0.1234	13.5618
18	119.1984	0.1122	40.7453	88.9525	0.1291	13.5413
19	119.1488	0.1135	40.6749	88.9566	0.1301	13.5236
20	119.1104	0.1134	40.6066	88.9452	0.1266	13.5440
21	118.9857	0.1183	40.4706	88.9345	0.1254	13.5735
22	**118.9819**	**0.1154**	**40.3918**	**88.9294**	**0.1240**	**13.6057**
23	119.0189	0.1125	40.3913	88.9247	0.1239	13.5618

Table 24: SVM Linear Regression RFE Performance Results - Cluster 1 Features

As shown in Table 24, the best performing configuration for Cluster 1 is with 22 features as this configuration exhibits the lowest RMSE value.

No.Feat.	RMSE	RS	MAE	R.SD	RSSD	M.SD
1	120.5461	0.1110	41.2564	88.5805	0.1345	13.5097
2	119.3357	0.1587	41.1663	88.6174	0.1983	13.4309
3	118.4193	0.1746	40.8625	88.7393	0.2014	13.4537
4	**118.1804**	**0.1591**	**40.4493**	**89.0153**	**0.1656**	**13.4863**
5	118.3473	0.1412	40.4472	89.0553	0.1449	13.4836
6	118.4919	0.1338	40.5085	89.0362	0.1425	13.4643
7	118.5247	0.1306	40.5262	89.0438	0.1397	13.4806
8	118.5308	0.1279	40.5351	89.0692	0.1339	13.4842
9	118.5572	0.1263	40.5746	89.0738	0.1326	13.4642
10	118.6313	0.1236	40.6037	89.0679	0.1305	13.4583
11	118.6471	0.1220	40.6101	89.0703	0.1294	13.4634

Table 25: SVM Linear Regression RFE Performance Results - Cluster 2 Features

Similarly, Table 25 shows that the best performing configuration for Cluster 2 is with four features.

No.Feat.	RMSE	RS	MAE	R.SD	RSSD	M.SD
1	121.9091	0.0703	41.3486	88.4323	0.1923	13.6225
2	121.8911	0.0477	41.3837	88.4288	0.1497	13.6168
3	121.8564	0.0471	41.4271	88.4173	0.1299	13.6195
4	121.8222	0.0568	41.4488	88.4510	0.1368	13.5859
5	**121.5831**	**0.0471**	**41.4591**	**88.5314**	**0.0741**	**13.5415**

Table 26: SVM Linear Regression RFE Performance Results - Cluster 3 Features

Finally, as depicted in Table 26, the best performing configuration for Cluster 3 is with five features.

Given the results above, the RFE analysis on the linear SVM model recommends the thirty-one features shown in Table 27.

Feature ID	Cluster
X805	1
X789	1
X709	1
X769	1
X644	1
X661	1
X760	1
X837	1
X763	1
X761	1
X751	1
X628	1
X815	1
X700	1
X16	1

Feature ID	Cluster
X23	1
X810	1
X771	1
X729	1
X17	1
X90	1
X25	1
X641	2
X841	2
X683	2
X680	2
X823	3
X844	3
X665	3
X793	3
X733	3

Table 27: SVM Linear Regression RFE Feature Details Based on Best Performing Fits

4.3 Regression Results Analysis

4.3.1 Linear Regression Results

The Linear Regression analysis produced straightforward coefficient performance measurements via ANOVA analysis. These measurements provide insight into the statistical significance of the predictors used during the fit of each model. This analysis was performed for each cluster and the results of these models can be seen in the tables that follow.

Coef.	Estimate	Std. Error	t-value	P-value
(Intercept)	39.2418	14.1503	2.7732	0.006
X133	−4.9187	10.7200	−0.4588	0.646
X709	**14.3685**	**2.9709**	**4.8364**	**< 0.001**

Table 28: Linear Regression Coeficient Significance - Cluster 1 Features

As shown in Table 28, only feature X709 is considered statistically significant for Cluster 1 when using a 95% confidence level.

Coef.	Estimate	Std. Error	t-value	P-value
(Intercept)	24.7377	7.4173	3.3352	< 0.001
X841	**23.7765**	**7.9464**	**2.9921**	**0.003**

Table 29: Linear Regression Coeficient Significance - Cluster 2 Features

Furthermore, for Cluster 2, Table 29 indicates the only predictor used in this cluster, X841, is statistically significant in the fit as tested to a confidence level of 95%.

Coef.	Estimate	Std. Error	t-value	P-value
(Intercept)	27.9214	10.2497	2.7241	0.007
X665	**12.4307**	**4.6351**	**2.6819**	**0.007**
X733	−18.0618	21.3940	−0.8442	0.399
X793	−0.8454	9.4082	−0.0899	0.928
X823	−0.8158	4.6748	−0.1745	0.862

Table 30: Linear Regression Coeficient Significance - Cluster 3 Features

Next, for Cluster 3, as shown in Table 30, the analysis finds that feature X665 is significant while the rest are not, using a confidence level of 95%.

Coef.	Estimate	Std. Error	t-value	P-value
(Intercept)	23.5552	15.9978	1.4724	0.141
X133	5.3615	12.5002	0.4289	0.668
X665	6.9752	4.7894	1.4564	0.146
X709	**13.3735**	**3.1846**	**4.1994**	**< 0.001**
X733	−29.3922	21.9972	−1.3362	0.182
X793	−6.2021	9.6254	−0.6444	0.520
X823	−3.8331	4.9221	−0.7787	0.436
X841	18.1972	10.0595	1.8090	0.071

Table 31: Linear Regression Coeficient Significance - All Features

Finally, the results for a regression fit across all features is shown in Table 31. The feature performance in this case is less desirable than the results of each cluster separately, as is typically the case for linear regression models (adding complexity can reduce the quality of the fit [36]). The only statistically significant predictor in this model is X709 while the remainder are not significant, using a confidence level of 95%.

A summary comparison of the performance of each of these models is depicted in Table 32 below. These results will be compared with the performance of the other regression models in Section 4.3.4.

Name	RMSE	RS	MAE	R.SD	RSSD	M.SD
lm-1	**117.6001**	**0.1029**	**51.4747**	**88.3889**	**0.1552**	**13.2767**
lm-2	118.5308	0.0449	52.8058	88.6165	0.0425	13.3494
lm-3	119.0524	0.0304	53.0665	89.0329	0.0392	13.5706
lm-all	117.3171	0.0984	50.9282	88.8316	0.1249	13.3957

Table 32: Linear Regression Performance Summary

4.3.2 Neural Network Results

Analysing the Neural Network results requires a comparison of strictly the RMSE and related performance variables across the relevant tuning arguments. Since this is a regression analysis, the activation function for the Neural Network output layer was set to linear. For each cluster, the Neural Network performance was gauged multiple times while varying the input parameters to the Feed Forward, Single Hidden Layer Neural Network model. These parameters are as follows:

- Size (S) - The number of hidden nodes in the Neural network: varying from 8 to 10 in steps of 1,
- Decay (D) - The rate of the weight decay: varying from 252 to 254 in steps of 1. Note: Decay refers to the rate of decay of the neuron weight penalty used in the calculation of the Neural Network cost function. This allows a variation in the trade off between error cost and large weight size.

The following Table 33 depicts the best performance of the Neural Network regression model against each cluster and for all features:

N.	S	D	RMSE	RS	MAE	R.SD	RSSD	M.SD
nn-1	9	253	117.00	0.07	49.03	88.74	0.05	13.66
nn-2	**10**	**252**	**115.37**	**0.12**	**48.92**	**89.57**	**0.08**	**13.19**
nn-3	8	252	118.41	0.04	51.66	88.94	0.04	13.55
nn-all	9	253	115.41	0.10	47.59	89.07	0.06	13.62

Table 33: Neural Network Regression Performance Summary

Given these results, the Neural Network performed best with Cluster 2 fea-

ture as well as on all features combined.

4.3.3 Linear Support Vector Machine Results

Analysing the Linear Kernel SVM performance is similar to that of the Neural Network analysis in that the SVM model requires tuning parameters to be varied and tested against the model inputs in order to determine best fit. This process again results in a table of tuning performance values for each cluster. The tuning value that is varied in the SVM configuration is the regularization parameter 'C'. For the purposes of this analysis, C is tuned from 0.0012 to 0.0013 in steps of 0.000001.

The best tuning performance for each cluster and for all features combined can be observed in Table 34 below:

N.	C	RMSE	RS	MAE	R.SD	RSSD	M.SD
sl-1	0.001 291	118.43	0.12	40.43	89.76	0.13	14.05
sl-2	0.001 289	**117.58**	**0.16**	**40.46**	**89.80**	**0.16**	**14.04**
sl-3	0.001 268	120.91	0.05	41.46	89.44	0.09	14.21
sl-all	0.001 202	118.35	0.11	40.49	89.89	0.14	13.99

Table 34: Linear SVM Regression Performance Summary

Given these results, the SVM model demonstrates similar but slightly poorer results as compared to the Neural Network model. A thorough comparison of each model is presented in the following section.

4.3.4 Summary of Regression Results

By selecting the best performing tuning setting from each of the cluster runs of each of the models, it is possible to sort the list by RMSE and determine the best performing model. As shown in Figure 11, the Neural Network model on Cluster 2 was the best performer.

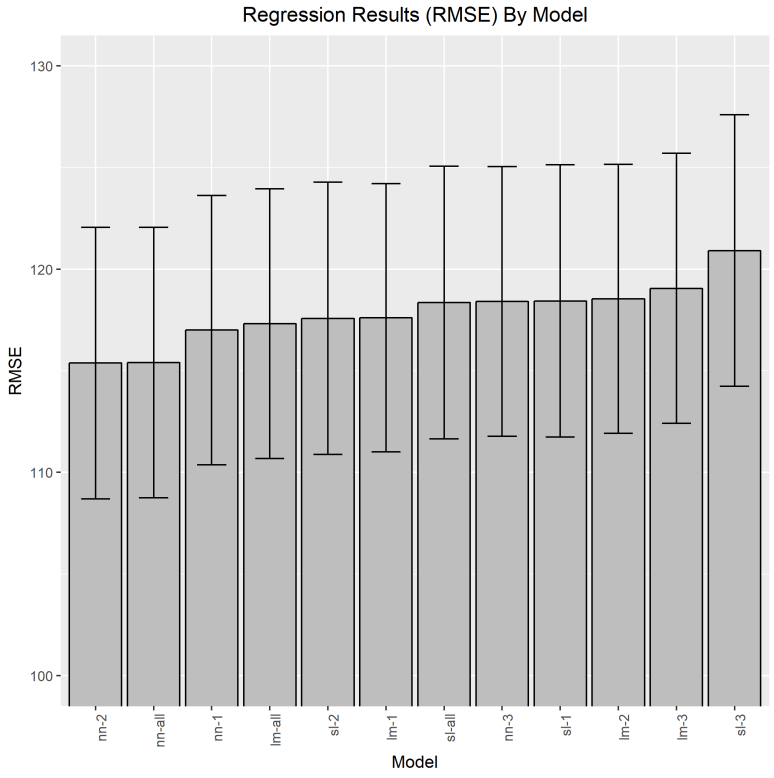

Figure 11: Graphcial Representation of the RMSE Performance of Each Regression Model

Although the Neural Network model outperformed the other models, it should be noted that all models demonstrated similar performance. When comparing their performance to the distribution of the software defect outcome space, it is obvious that none of these models are significantly impor-

tant.

This can be further expanded when analysing the distribution of the software defect counts column which is being used as the outcome dependent variable for fitting these models.

Measure	**Value**
Min.	0
1st Qu.	0
Median	5
Mean	40.2182
3rd Qu.	26
Max.	3097

Table 35: Software Defect Outcomes Count Variable Statistics

Table 35 depicts the distribution of the software defect counts per software application. As shown, some software applications have exhibited 0 software defects. Additionally, the First Quartile, or the Mean of the Minimum and Median of the distribution, is also 0, indicating that there is a non-trivial number of software applications with a defect count of zero. Said another way, at least 25% of the software applications have 0 defects. Next, the Median is 5, indicating that the center of the distribution is skewed lower than the Mean of the distribution. The Mean of approximately 40 demonstrates a significant gap between the Median and Mean, thus confirming this observation of skew. The Third Quartile of approximately 26, which indicates that 25% of the software applications have at least 26 software defects. Finally, at least one software application has a Maximum number of software defects of 3097.

Analysing Table 35 further, the Mean of the defect distribution is approx-

imately 40 while the best performing regression model had an RMSE of approximately 115 (Neural Network) and a Mean Absolute Error of 40.4 (SVM). Given that the RMSE of the best model is almost 3 times that of the mean of the distribution of the outcomes and the MAE is approximately equal to the mean of the distribution, the regression models are invalid and can be rejected.

4.4 Binary Classification Results Analysis

Binary Classification involves modifying the prediction column into a factor instead of an integer value. The factors considered in this section are as follows:

- Has SPR: The defect count for a given application is greater than 0.
- No SPR: The defect count for a given application is 0

Using the definitions above, the output column was transformed into a binary factor column. The distribution of this classification can be seen in Table 36.

Class	Occurrence	Percentage
HAS_SPR	576	74.81%
NO_SPR	194	25.19%

Table 36: Binary Classification Outcome Distribution Statistics

It is apparent that there is a class imbalance in favour of applications having a software defect. Up-sampling will be used in the following analysis in order to compensate for this imbalance.

4.4.1 Decision Tree Results

The Decision Tree model used in this analysis is the Classification and Regression Tree (CART) and is provided by the rpart library in R [51]. This particular Decision Tree accepts a tuning parameter of cost complexity (CP). The values used for this analysis are in the range of 0.006 to 0.04 in steps of 0.001. The model was developed multiple times using each of these parameters with all features as well as with all features from each feature cluster.

Table 37 shows the best performing model for each cluster and for all of the features:

Name	CP	AUC	Precision	Recall	F
rp-1	0.0090	0.7738	0.8427	0.6157	0.7083
rp-2	0.0140	0.7598	0.8291	0.4960	0.6169
rp-3	0.0100	0.7433	0.7913	0.4570	0.5724
rp-all	**0.0110**	**0.7835**	**0.8505**	**0.6074**	**0.7052**

Table 37: Binary Classification - Decision Tree Performance Summary by Model

Given the AUC results above, the Decision Tree model performed best using all features. While the F score of the model using all features is slightly less than that of the Cluster 1 model, the trade-off of higher Precision in the all features model is the deciding factor.

4.4.2 Random Forest Results

The Random Forest model used in this analysis requires a tuning parameter of mtry. This parameter defines the Minimum feature count used to TRY and grow a tree in a Random Forest. This parameter was varied from 0 to 12 in steps of 1. Similar to models above, this model was executed against each feature Cluster and then against the entire feature list. Table 38 shows the best result for each run:

Name	mtry	AUC	Precision	Recall	F
rf-1	**2**	**0.8636**	**0.8216**	**0.7482**	**0.7818**
rf-2	1	0.8121	0.8150	0.5771	0.6742
rf-3	4	0.7825	0.7888	0.6975	0.7393
rf-all	9	0.8630	0.8092	0.7942	0.8004

Table 38: Binary Classification - Random Forest Performance Summary by Model

The Random Forest demonstrates similar positive performance for both Cluster 1 and all features. In this case, however, the model executed on Cluster 1 features demonstrates better results given the AUC and Precision together.

4.4.3 Neural Network Results

The activation function for the Neural Networks used in this Binary Classification analysis was sigmoidal. This Neural Network is a single hidden layer Feed Forward network. The model used in this analysis accepts tuning parameters of Size and Decay. These parameters were varied as follows:

- Size (S) - 1, 3, 5, 7, 9, 11, 13, 15, 17, 19
- Decay (D) - Varying from 0.05 to 0.5 in steps of 0.01

Refer to Table 39 for the results of the best execution from each run of the Neural Network model:

Name	S	D	AUC	Precision	Recall	F
nn-1	7	0.45	0.8550	0.8294	0.6475	0.7254
nn-2	1	0.38	0.8195	0.8319	0.4698	0.5980
nn-3	11	0.19	0.7789	0.7785	0.5555	0.6458
nn-all	**19**	**0.50**	**0.8609**	**0.8261**	**0.7096**	**0.7617**

Table 39: Binary Classification - Neural Network Performance Summary by Model

As shown, the Neural Network performance is also best for Cluster 1 and all features, however, all features performs best given the AUC, Recall, and F score together.

4.4.4 Linear Support Vector Machine Results

The Linear SVM requires tuning parameters to be varied and tested against the model inputs in order to determine best fit. This process again results in a table of tuning performance values for each Cluster. The tuning value that is varied in the SVM configuration is the regularization parameter 'C'. For the purposes of this analysis, C is varied from 2 to 5 in steps of 0.01.

Refer to Table 40 for the results of the best execution from each run of the SVM model:

Name	C	AUC	Precision	Recall	F
sl-1	**3.16**	**0.8694**	**0.8903**	**0.5450**	**0.6735**
sl-2	3.20	0.8110	0.8308	0.3586	0.4985
sl-3	3.86	0.7598	0.7733	0.2238	0.3199
sl-all	2.74	0.8634	0.8835	0.5353	0.6646

Table 40: Binary Classification - SVM Performance Summary by Model

In a similar fashion as the other Binary Classification models, the SVM model demonstrates best performance on Cluster 1 features as well as all features together. It should be noted here however, that the Recall of all SVM models is much lower than the Decision Trees, Neural Networks and Random Forest models.

4.4.5 Summary of Binary Classification Results

Given the performance of each model above, the overall performance is plotted in Figure 12 below.

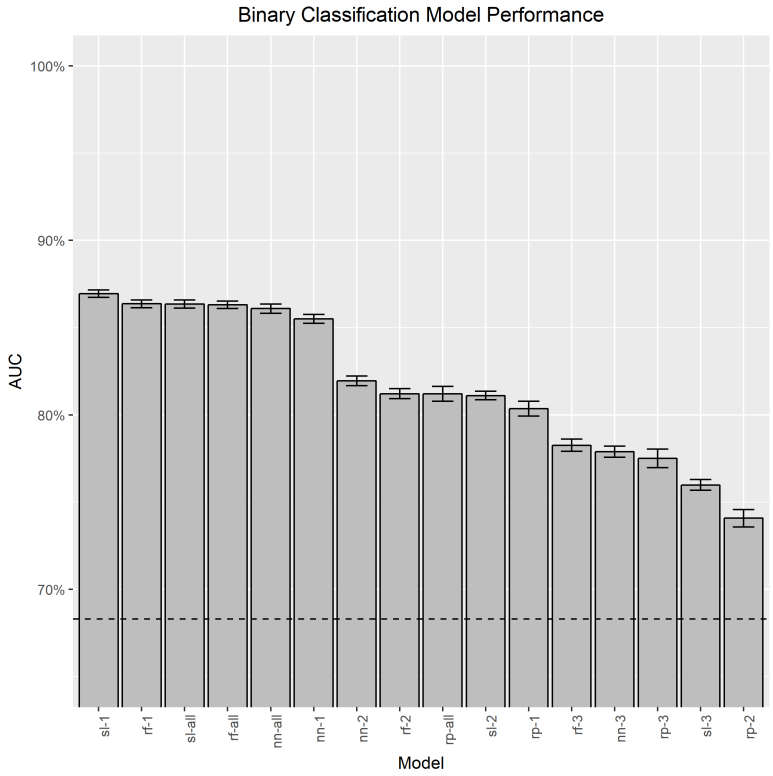

Figure 12: Graphical Performance Comparisons of Each Binary Classification Model

It is apparent that all of the models succeeded in providing an acceptable AUC greater than the threshold of 0.683, with the SVM model demonstrating the best performance for Cluster 1. The Random Forest model on Cluster 1 demonstrates the next best AUC value.

Table 41 depicts the averaged confusion matrix for the SVM model on Cluster 1 features based on a 10-fold cross-validated model analysis. It can be observed that the SVM model makes a large portion of its errors through false positives. The false negative rate on the other hand is much smaller.

		Predicted	
		Has SPR	No SPR
Actual	Has SPR	40.7662	5.0909
	No SPR	34.0390	20.1039

Table 41: Cluster 1 Features Set Confusion Matrix for the SVM Binary Classification Model

Further analysis of the SVM model provides an importance ranking of each of the features used during the model fitting. Figure 13 depicts the variable importance of the SVM model for Cluster 1:

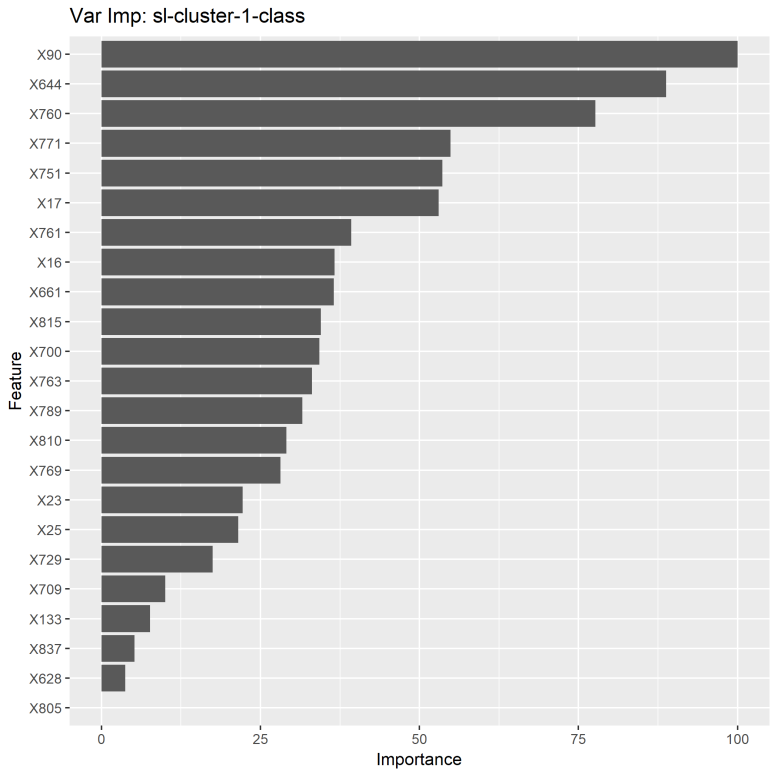

Figure 13: Cluster 1 Features Set Variable Importance in Fitting the SVM Binary Classification Model

Taking a closer look at the features with an predictive accuracy of greater

than 50%, Table 42 depicts these features as they map to SonarQube rules. These results indicate features that, according to the SVM Model, are strongly correlated with the presence of software defects in software applications.

Feature ID	**Description**	**Importance**
X90	Duplicated lines (Percentage)	100%
X644	Loggers should be private static final and should share a naming convention	88.77%
X760	Expressions should not be too complex	77.63%
X771	Unused method parameters should be removed	54.88%
X751	Local variables should not shadow class fields	53.61%
X17	Public documented API (Percentage)	53.02%

Table 42: SVM Binary Classification Important Features Mapping to SonarQube Rules

Table 43 depicts the averaged confusion matrix for the Random Forest model on Cluster 1 features based on a 10-fold cross-validated model analysis. This model has better performance in false positives than false negatives. The rate of false negatives is almost split at 50%.

		Predicted	
		Has SPR	No SPR
Actual	Has SPR	55.9610	12.2078
	No SPR	18.8442	12.9870

Table 43: Cluster 1 Features Set Confusion Matrix for the Random Forest Binary Classification Model

Similarly, Figure 14 depicts the variable importance of the Random Forest model on Cluster 1 features:

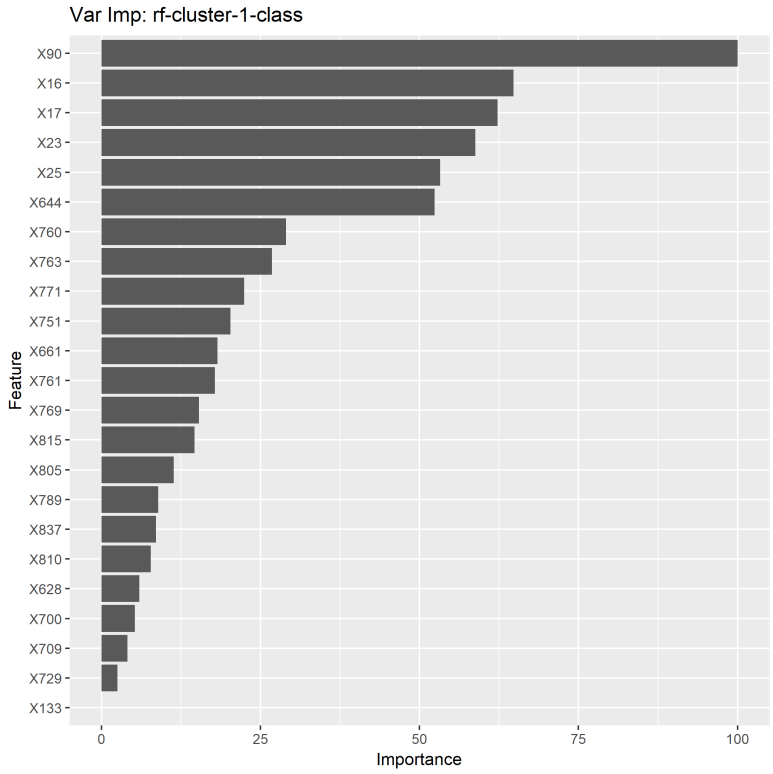

Figure 14: Cluster 1 Features Set Variable Importance in Fitting the Random Forest Binary Classification Model

Again, Table 44 shows the mapping features with a predictive accuracy of greater than 50% to their corresponding SonarQube rules. These results indicate features that, according to the Random Forest Model, are strongly correlated with the presence of software defects in software applications.

Feature ID	Description	Importance
X90	Duplicated lines (Percentage)	100%
X16	Comments (Percentage)	64.79%
X17	Public documented API (Percentage)	62.26%
X23	Complexity class	58.77%
X25	Complexity function	53.25%
X644	Loggers should be private static final and should share a naming convention	52.35%

Table 44: Random Forest Binary Classification Cluster 1 Features Set Important Features Mapping to SonarQube Rules

4.5 Multi-class Classification Results

Multi-Class Classification involves modifying the prediction column into a set of factors instead of an integer value. The factors considered in this section are as follows:

- No SPR: The defect count for a given application is 0
- Minimal SPR: The defect count for a given application is less than or equal to 4
- Moderate SPR: The defect count for a given application is less than or equal to 25
- High SPR: The defect count for a given application is less than or equal to 500
- Extreme SPR: The defect count for a given application is greater than 500.

Using the definitions above, the output column was transformed into a mul-

tiple factor column. The distribution of this classification can be seen in Table 45.

Class	Occurrence	Percentage
EXTREME˙SPR	7	0.91%
HIGH_SPR	189	24.55%
MINIMAL_SPR	185	24.03%
MODERATE_SPR	195	25.32%
NO_SPR	194	25.19%

Table 45: Multi-class Outcome Distribution Statistics

It is shown that there is a reasonable class balance this time, as the ranges for the classification buckets were chosen carefully to ensure a fair distribution. This distribution is applicable to similar domain and quality threshold software development programs across language types. If a different domain is analysed, the ranges chosen above will need to be revisited. Given this distribution of classes, up-sampling is still required for the Multi-Class Classification analysis.

4.5.1 Decision Tree Results

The CART Decision Tree model was used in this analysis as was used in the binary classification analysis. The CP values used for this analysis are in the range of 0.005 to 0.012 in steps of 0.0001. The model was executed multiple times against each of these parameters as well as against each feature Cluster, including against a pass with all features included.

Table 46 lists the best performing model fit for each Cluster and for all features.

Name	CP	AUC	Precision	Recall	F
rp-1	0.0072	0.6199	0.3097	0.3271	0.3148
rp-2	0.0058	0.5597	0.2780	0.2780	0.4240
rp-3	0.0107	0.5575	0.2761	0.3026	0.2685
rp-all	**0.0115**	**0.6342**	**0.3172**	**0.3427**	**0.3268**

Table 46: Multi-Class Classification - Decision Tree Performance Summary by Model

Table 46 demonstrates that while the performance was best on Cluster 1 features and all features, as was the case in Binary Classification. However, the overall AUC performance is much lower for Multi-Class Classification than it was for Binary Classification. The Decision Tree demonstrates the best performance in this case on all features with the highest AUC.

4.5.2 Random Forest Results

The same Random Forest model was used in this analysis as was used in the Binary Classification analysis. The mtry parameter was varied from 9 to 13 in steps of 1. Similar to models above, this model was executed against each feature Cluster and then against the entire feature list. Table 47 shows the best result for each run:

Name	mtry	AUC	Precision	Recall	F
rf-1	9	0.6723	0.3325	0.3084	0.3200
rf-2	9	0.6301	0.3007	0.2999	0.3003
rf-3	12	0.6076	0.2860	0.2645	0.3010
rf-all	**11**	**0.7040**	**0.3696**	**0.3393**	**0.3538**

Table 47: Multi-Class Classification - Random Forest Performance Summary by Model

The Random Forest model returns more acceptable results than the Decision Tree models with the rf-all model performing with an AUC of over 0.7. These results will be compared with the other model performances in the summary section presented later.

4.5.3 Neural Network Results

A sigmoidal activation function was used for the Neural Networks in this Multi-Class Classification analysis. This Neural Network is a single hidden layer Feed Forward network with a separate output for each class. The Neural Network model used in this analysis accepts tuning parameters of Size and Decay. These parameters were varied as follows:

- Size (S) - Varying from 4 to 17 in steps of 1
- Decay (D) - Varying from 0.05 to 0.5 in steps of 0.01

Refer to Table 48 for the results of the best execution from each run of the Neural Network model:

Name	S	D	AUC	Precision	Recall	F
nn-1	6	0.4500	0.6635	0.2966	0.2999	0.3293
nn-2	4	0.4700	0.6412	0.2972	0.4075	0.3322
nn-3	16	0.1000	0.5646	0.2485	0.2531	0.2868
nn-all	**4**	**0.3300**	**0.6866**	**0.3166**	**0.3025**	**0.3375**

Table 48: Multi-Class Classification - Neural Network Performance Summary by Model

As shown, the Neural Network models did outperform the Decision Tree models, however, they did not perform as well as the Random Forest models.

The model based on all features performed best, followed closely by the features of Cluster 1, as indicated by the AUC of each.

4.5.4 Linear Support Vector Machine Results

The Linear SVM used for the Multi-Class Classification utilized the tuning parameter, C, which was varied from 2 to 5 in steps of 0.01.

Refer to Table 49 for the results of the best execution from each run of the SVM models:

Name	C	AUC	Precision	Recall	F
sl-1	3.51	0.6084	0.2960	0.2353	0.2342
sl-2	2.30	0.5898	0.2259	0.2538	0.2391
sl-3	3.40	0.5631	0.2391	0.2556	0.2471
sl-all	**2.08**	**0.6199**	**0.3358**	**0.2550**	**0.3155**

Table 49: Multi-Class Classification - SVM Performance Summary by Model

While the SVM models outperformed the Decision Tree models slightly, they did not perform as well as the Neural Network or Random Forest models. Again, all features and Cluster 1 features were the best performers.

4.5.5 Summary of Multi-Class Classification Results

Each model performance is captured in the comparison chart, Figure 15. The results of this chart are ordered from best to worst performance. The dashed line across the chart indicates the minimum AUC possible, as a reference threshold.

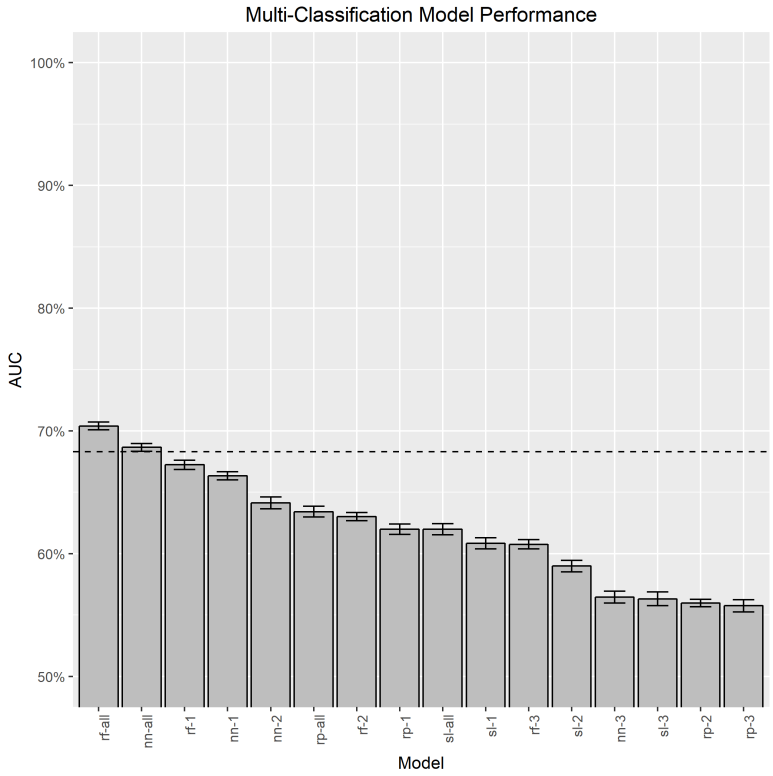

Figure 15: Graphical Performance Comparisons of each Multi-Class Classification Model

These results imply that for the Multi-Class Classification analysis, the Random Forest model executed over the entire feature set was the best performing prediction model followed by the Neural Network model executed over the entire feature set as well. These two models were the only models to successfully outperform the performance threshold AUC of 0.683.

Table 50 depicts the averaged confusion matrix for the Random Forest model on all features based on a 10-fold cross-validated model analysis. This matrix shows that the Random Forest Model is able to properly classify High and None occurrences best with Moderate and Minimal next best. Overall, the

error cases appear to straddle the transition classes (High and Minimal for a Moderate prediction for example).

		Predicted				
		Extreme	High	Moderate	Minimal	None
Actual	Extreme	0.0000	2.1169	2.3377	0.7532	2.2078
	High	0.6234	11.8571	5.1039	2.9740	4.4675
	Moderate	0.1558	5.0779	9.8571	5.2338	2.5065
	Minimal	0.0000	1.4545	5.5325	9.3636	5.2208
	None	0.1299	4.0390	2.4935	5.7013	10.7922

Table 50: All Features Confusion Matrix for the Random Forest Multi-Class Classification Model

Further exploration of the Random Forest model demonstrates the importance ranking of each of the features used during the model fitting. Figure 16 depicts the variable importance of the Random Forest model for all features:

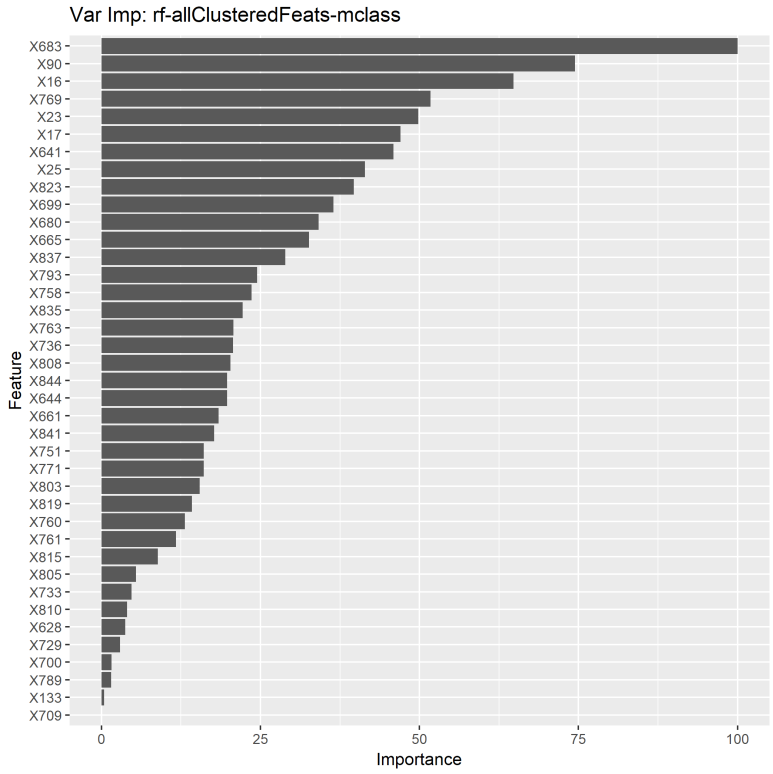

Figure 16: All Features Variable Importance in Fitting the Random Forest Multi-Class Classification Model

Taking a closer look at the features with a predictive accuracy of greater than 50%, Table 51 shows these features as they map to SonarQube:

Feature ID	**Description**	**Importance**
X683	Avoid commented-out lines of code	100%
X90	Duplicated lines (Percentage)	74.47%
X16	Comments (Percentage)	64.75%
X769	Constants should be declared final static rather than merely final	51.75%

Table 51: Random Forest Multi-Class Classification Important Features Mapping to SonarQube Rules

Table 52 depicts the averaged confusion matrix for the Neural Network model

on all features based on a 10-fold cross-validated model analysis. This matrix shows that the model is best at predicting a High defect occurrence rate with None coming in a close second. Unfortunately, the model does not perform very well on Minimal or Moderate with Extreme having the worst performance. In a similar fashion to the Random Forest Model, the Neural Network Model makes most incorrect predictions along the transition between nearby classifications.

		Predicted				
		Extreme	High	Moderate	Minimal	None
Actual	Extreme	0.0649	2.5974	2.6234	0.8052	2.9610
	High	0.3636	10.5974	4.7273	3.5714	4.0260
	Moderate	0.0649	4.4935	6.8961	5.2338	2.3117
	Minimal	0.0390	2.6753	6.8571	8.9351	6.8701
	None	0.3766	4.1818	4.2208	5.4805	9.0260

Table 52: All Features Confusion Matrix for the Neural Network Multi-Class Classification Model

Similarly, Figure 17 depicts the variable importance of the Neural Network model for all features:

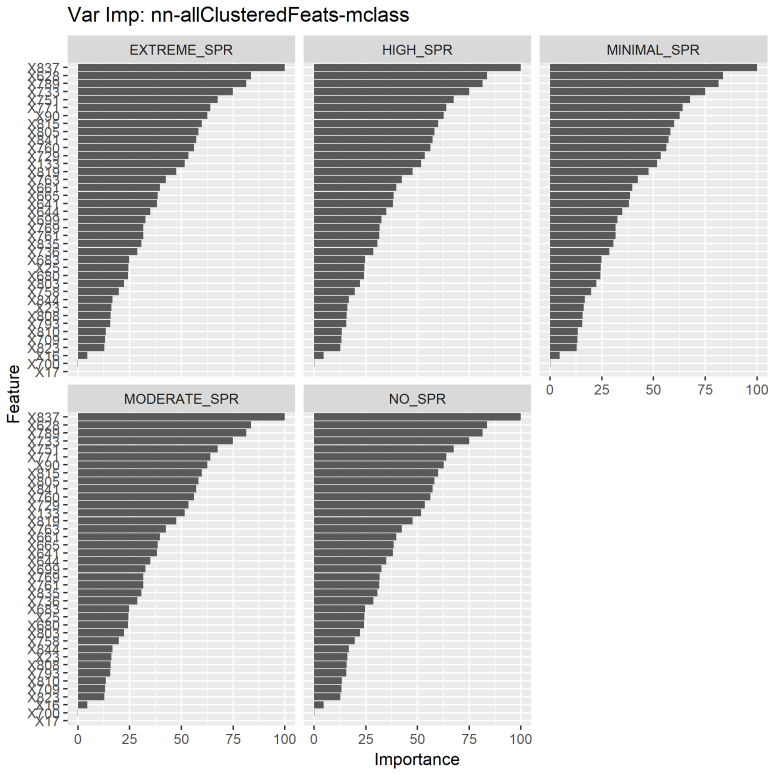

Figure 17: All Features Variable Importance in Fitting the Neural Network Multi-Class Classification Model

Again, Table 53 shows the features with a predictive accuracy of greater than 50% as they map to their corresponding SonarQube rules:

Feature ID	Description	Importance
X837	Deprecated code should be removed eventually	100%
X628	equals(Object obj) and hashCode() should be overridden in pairs	83.63%
X789	Return of boolean expressions should not be wrapped into an if-then-else statement	81.41%
X733	Method names should comply with a naming convention	74.98%
X751	Local variables should not shadow class fields	67.58%
X771	Unused method parameters should be removed	64.01%
X90	Duplicated lines (Percentage)	62.59%
X815	Nested blocks of code should not be left empty	59.94%
X805	A field should not duplicate the name of its containing class	58.24%
X841	TODO tags should be handled	57.25%
X760	Expressions should not be too complex	56.18%
X729	Empty arrays and collections should be returned instead of null	53.46%
X133	SQALE Rating	51.67%

Table 53: Neural Network Multi-Class Classification Important Features Mapping to SonarQube Rules

5 Discussion

5.1 Feature Elimination

The cluster analysis shows that most of the features easily group into two main categories, with five independent features left in their own separate clusters. It would not have been relevant to perform fitting on the individual clusters on their own, as they would not contain sufficient information in order to provide significant insight by themselves. Instead, the individual features were combined into another cluster, with the intent that this disparate cluster would provide useful insights. Unfortunately it did not, as in most cases, Cluster 3 performed the poorest in the later analysis. It is likely that Cluster 3 performed poorly as it consisted of a collection of single features that each had no relationship with any of the other features in the problem space. This behaviour implies that these features have little importance in predicting the outcome of software defects. It is also interesting to note that, in most cases, the Features in Cluster 1 performed the best, which indicates that this cluster is most strongly correlated with the prediction of software defect rates.

The RFE analysis discarded a large number of features for the regression analysis, with the goal that this would lead to a higher performing regression model. As the regression analysis indicated, none of the models were able to achieve an acceptable RMSE level. The Neural Network model performed on Cluster 2 features provided the smallest RMSE value of 115.2162, but even this value was almost three-times that of the mean of of the outcome variable

(40).

5.2 Regression

Due to the large discrepancy between RMSE and MAE and the mean, a model that always chooses the mean value of the outcome variable would demonstrate a smaller RMSE and a similar MAE than the regression models fitted in this analysis. Clearly, the data in the features analysed are not reasonably correlated with the outcome variable when expressed in a continuous data space through regression.

Regression demonstrated very poor performance across all models tested. This should come as little surprise given the complexity of the data set and high variability in the results. Additional research can still be conducted in this area, however, as more data is collected and more features are analysed, it is possible that some rules and metrics do exhibit a linear relationship with the outcomes and that a more favourable fit can be found if the right data is analysed. Additionally, the regression analysis could be re-run with a set of non-linear transforms applied to the features, as the relationship between the features and the outcome are likely non-linear given the results of the fitting in this result. Some examples of non-linear transforms are: Box Cox, Yeo Johnson, Spatial Sign, as well as many others [37].

5.3 Binary Classification

The binary classification analysis provided much more promising results, with the SVM and Random Forest algorithms proving most effective at predicting the software defect outcomes. All of the models in this analysis provided results that outperformed the performance threshold of 0.683, with the SVM model, executed over the Cluster 1 feature set, demonstrating an AUC of 0.8694. This means that a model can be built that uses a minimal feature set of less than ten SonarQube metrics and rules.

Binary Classification performance suffered significantly from the class imbalance observed between the software applications. While up-sampling was performed in an attempt to compensate for this issue, the overall performance of the fit of the models was negatively impacted as a result as shown in the confusion matrices in Tables 41 and 43. Notwithstanding this, the Binary Classification analysis produced a moderately successful model for defect prediction, and also helped narrow the focus of the SonarQube features to a very manageable list of twelve of the most important metrics and rules to monitor for defect management, as shown in Tables 42 and 44. A software team could monitor this small list of metrics and rules and use it as a guide to manage a set of actions to control the measurements and mitigate their impact on the generation of software defects in the source code.

5.4 Multi-Class Classification

The Multi-Class Classification analysis provided less promising results than the Binary Classification data above, with the Random Forest and Neural Network algorithms proving to be the only models that could outperform the AUC threshold of 0.683.

Multi-Class Classification performed poorer than Binary Classification, as shown in the confusion matrices in Tables 50 and 52, and this poor performance can be attributed to the additional complexity introduced in the Multi-Class Classification problem. It is much more reliable to predict the presence or absence of a software defect than it is to predict the specific defect rate range. These results are in line with the poor results observed in the regression results as well.

5.5 Implications of Findings

Both of the classification models provided additional insight into the level of importance of the SonarQube metrics and rules, ultimately leading the results shown in Table 54, a list of important features, ranked from most important to least important.

Feature ID	Description	Importance
X683	Avoid commented-out lines of code	100%
X837	Deprecated code should be removed eventually	100%
X628	equals(Object obj) and hashCode() should be overridden in pairs	83.63%
X789	Return of boolean expressions should not be wrapped into an if-then-else statement	81.41%
X733	Method names should comply with a naming convention	74.98%
X90	Duplicated lines (Percentage)	74.47%
X751	Local variables should not shadow class fields	67.58%
X16	Comments (Percentage)	64.75%
X771	Unused method parameters should be removed	64.01%
X815	Nested blocks of code should not be left empty	59.94%
X805	A field should not duplicate the name of its containing class	58.24%
X841	TODO tags should be handled	57.25%
X760	Expressions should not be too complex	56.18%
X729	Empty arrays and collections should be returned instead of null	53.46%
X769	Constants should be declared final static rather than merely final	51.75%
X133	SQALE Rating	51.67%

Table 54: Details of All Significant Features as Determined by Analysis

The list of important features shown above contain both expected and surprising results. The expected results include a feature such as X760 ("Expressions should not be too complex") as it would stand to reason that the more complex a piece of software is, the higher the probability that it will contain defects.

The surprising results in this list include: X683 ("Avoid commented-out lines of code"), X837 ("Deprecated code should be removed eventually"), X90 ("Duplicated lines"), and X16 ("Comments"). "Duplicated lines" is of particular interest in that it highlights code that may be designed poorly, perhaps by not capitalizing on Object Oriented methodologies, software may exhibit high counts of the "Duplicated lines" metric. "Avoid commented-out lines of code" is another interesting example as on their own, commented out lines of code should actually have no impact on how well a software application runs, but clearly, there is an impact as shown here. "Comments" demonstrates that code that is not well commented may exhibit higher defect rates, which could imply that due to the lack of comments, developers may not completely understand how the code is working. Additionally, "Deprecated code should be removed eventually" may be an indicator of general bad coding practice given that the deprecated code has not yet been cleaned up.

Features at the bottom of this list include: X815 ("Nested blocks of code should not be left empty"), X729 ("Empty arrays and collections should be returned instead of null"), and X769 ("Constants should be declared final static rather than merely final"). These features provide a predictive accuracy of marginally greater than 50%, so although they remain at the bottom of this list, they should still be considered significant. Of interest is "Nested blocks of code should not be left empty" as it appears to be a minor variant of "Avoid commented-out lines of code". The same reasoning behind "Avoid commented-out lines of code" and "Duplicated lines" for their importance

relates to “Nested blocks of code should not be left empty”: These violations indicate a more systemic problem with bad coding practice.

6 Conclusions and Recommendations

6.1 Summary

The source code analysed in this research was obtained from three major software development programs. The software application source code for all three programs was stored in an SVN version control system. Due to the use of SVN, it was possible to obtain a historical snapshot of each of the source code repositories at the desired phase of the corresponding software development program. In each case, the code snapshot was taken after major software development had concluded, but prior to the start of formal software testing. This was done in order to obtain a snapshot of code that had the highest potential for latent defect content prior to defect correction.

In order to ensure consistency and comparable results, the metrics and rules sets were generated using the same baseline of SonarQube. In each case, SonarQube was used to perform static analysis of the chosen software source code snapshots. The resultant data was stored in an SQL database and then exported into CSV files for analysis. This process was performed over the course of several months, as the analysis of such large sets of code was computer resource intensive. Additionally, several executions of the analysis were performed in order to ensure consistency in the generated results.

The software defect data for each program was exported from the corresponding software defect repository. The data format for each of the programs differed slightly as each program had been executed in different years

by different teams, and thus the reporting techniques had small variations. Significant manual effort was spent in reconciling these differences in order to produce a unified data set. This effort was necessary in order to allow for an analysis of all programs simultaneously.

The static analysis data and software defect data was then joined and aggregated. The aggregation performed was a simple mapping of software defect occurrence count per unique software application. This aggregation was then joined with the table that contained each static analysis measure for each software application. The resulting data was a set of independent variables: the static analysis metrics and rules, and the dependent variable: the software defect measurements.

As indicated in the Literature Review, the issue of Metrics Galore became apparent when the entire problem space was analysed. There were over 150 different Static Analysis metrics and rules generated by the SonarQube tool, and analysing every one would have been nearly impossible. In addition, it would not have been fruitful to blindly attempt to fit a regression or classification model against the entire feature set, as it would have most likely either lead to an over-fitted model that would fail when faced with new data or simply not result in a viable fit at all. This issue necessitated the use of various feature elimination techniques in order to narrow the feature set down to a more manageable set of variables. This was done using zero and near-zero variance analysis, inter-correlation analysis as well as recursive feature elimination. After each of these methods were applied, the feature set reduced correspondingly.

Prior to performing modelling analysis on the reduced feature set, one additional data preparation process was applied: clustering. Clustering was performed in order to group similar features together, thus allowing for several models to be fitted for each algorithm type. The motivation behind this step was to avoid pairing dissimilar features together that could negatively influence the outcome of a model fitting attempt. By grouping similar features together and then running model fits on each cluster separately, and then again as a whole, it is possible to rule out the negative impact of such dissimilar features.

Various regression algorithms were applied to the data set, with similarly poor results. None of the models used in this analysis demonstrated a statistically significant performance in regression. This is unsurprising given the relatively narrow scope of variables being considered for software defect outcomes.

By stratifying the software defect count variable into different groupings, it was possible to perform a classification analysis. First, by looking at whether a particular software application had at least one defect or not, Binary Classification could be performed. Similarly, by creating bins for software defect amounts, a Multi-Class Classification could be performed. In the case of the Binary Classification analysis, due to the disparity in numbers of applications with defects versus applications without defects, a class imbalance was present. In order to address this class imbalance, upsampling of the data points was performed prior to fitting the models. The class imbalance problem could be avoided in Multi-Class Classification as the

groupings were selected in order to create a fair distribution across classes. Similar distributions will have to be customized for the target analysis based on the performance of the software defect count values.

Finally, given the favourable results of the Classification research, and due to the nature of the Classification models used, it was possible to extract an ordered list of important features from the most successful models. This was possible, in particular in the SVM, Neural Network and Random Forest analysis, due to the fact that the models inherently rank the most influential features internally. Developing the method to generate the ranked list of important features is considered the most valuable finding in this research, as it can be directly applied to positively impact the focus of a software development team today, by guiding them to manage the features that matter most for software quality.

6.2 Conclusions

The results from this study were interesting and provide helpful insight into the software defect prediction problem. The implications of this research will help software development teams to focus their efforts on a manageable list of software metrics and rules that can be used to mitigate the impact of latent software defects.

During the course of this research, it was found that the given data set had no significant correlations that could be explained by a regression model. This is an expected finding given the complexities of software development and the

relatively narrow scope of this research as compared to the countless variables that affect software development quality. Nonetheless, Binary and Multi-Class classification models were able to demonstrate a reasonably significant correlation between a certain subset of features and the prediction of software defects.

Using the results of the Binary and Multi-Class Classification analysis, it was possible to derive the subset of features that play the largest role in predicting software defect outcomes. This list, Table 54, contains a mix of expected and unexpected rule violations and metric values. The key important takeaway from the subset of features is that the list is small enough to be adequately managed by a software team during a typical development cycle. This research has succeeded in providing a method for software development teams to focus on a small set of critical measurements that, if carefully managed, will have a positive impact in the quality of the code that the team produces.

Given the findings in this research, a reasoning that could be used to explain the results is that important features may serve as warning signs to software developers, team members, peers, and managers. This suggests that the developers working on a particular software application that exhibit a high number of the code malpractices listed in Table 54 may not be adequately skilled for the task at hand. It should be noted that these metrics and rules should not simply be treated as indications of problems that should be fixed, but instead, they should be looked upon as a "canary in the coal mine". If an application demonstrates a large number of violations of the metrics or

rules in Table 54, then action must be taken to carefully review contributor's code. More experienced developers should provide mentor-ship and guidance to the developers who have introduced these violations as they indicate a lack of skill or understanding of the problem at hand, the coding language being used or a combination of any number of these or other factors.

It is expected that by using these results, a team of software developers will be able to monitor, manage and control the measurements of the recommended feature focus area items during regular software development. This effort would fit in well with an Agile software development process, and could be made a part of the retrospective analysis in a Scrum sprint. By doing this, it is anticipated that software teams will experience fewer defects during acceptance testing by being proactive in addressing the most likely indicators of latent defects early, thus reducing potential cost and schedule impacts to the development programs.

It should be noted that due to the removal of undetected features in this analysis, the results and recommendations in this study are most applicable to similar software development programs: Large Java-based data processing and user entry applications without a web or mobile component. If web, mobile or other types of development programs must be analysed, the models developed in this research must be retrained as new metric and rule measurements may provide valuable insight that have not been considered here.

Finally, software development management teams will benefit from the predictive models by using them to gauge the health and status of ongoing

software development programs. While the results of these models cannot predict an exact software defect measurement, they can predict a category of software defect measurements and this information will serve to aid managers to plan budgets, schedules and resources accordingly as well as enact mitigation plans if deemed necessary.

Based on these findings, the following conclusions can be drawn based on the goals of the thesis at the outset:

1. Correlation
    - Goal: Demonstrate a statistically significant correlation between static analysis metrics and rules and software defects.
    - Result: A statistically significant correlation has been demonstrated with both Binary and Multi-Class Classification. This objective has been achieved.
2. Machine Learning Model
    - Goal: Produce a machine learning model that can be used to accurately predict software defects.
    - Result: Several reliable classification models have been generated that can apply to similar domains and purposes. Since the regression analysis did not produce statistically significant models, software defect counts cannot be accurately predicted as a number but instead of a range of values. This objective has only been partially achieved.

3. Manageable List

    - Goal: Identify the 10 most significant static analysis metrics and rules that attribute to the prediction of the software defects.
    - Result: A list of 16 of the most significant static analysis metrics and rules. This objective has been achieved.

4. Assessments

    - Goal: Provide an assessment of the results and recommendations for future software development programs.
    - Result: The important conclusions drawn from the unexpected results will serve to be very valuable to software development efforts in the future. This objective has been achieved.

6.3 Recommendations

Despite all of the results and analysis performed herein, there is still a lot more that could be done in this field to improve software defect estimation and mitigation. Below is a list of future work, in order of decreasing priority:

1. Re-run the Multi-Class Classification analysis with a rank-based performance optimization given the ordinal nature of the classifications. This type of analysis should boost accuracy for software applications that fall near the boundary line between two adjacent classifications. This would help prevent a Minimal occurrence classification from being

incorrectly classified as extreme, since the two classes are not adjacent with each other. Likewise, an incorrect classification between two adjacent classes in this analysis would be less critical than an incorrect classification across classes that are not adjacent with each other.

2. Supplement the Linear Regression ANOVA analysis with an intra-feature performance analysis in order to determine feature combinations that provide valuable insight.

3. Revisit near-zero variance feature elimination in order to determine if any of these features could add value to the models [52].

4. Analyse the features eliminated due to collinearity as they may provide additional insight into the behaviour of the software defects through feature interaction.

5. Obtain more data from additional software development programs.

6. Only SonarQube and FindBugs rules and metrics were analysed in this research, by adding more rule sets into SonarQube and performing the analysis again, the model could be made more effective [32].

7. Perform a time-series analysis of metrics, rules, and defects relationship by taking multiple snapshots of each of these repositories.

8. Attempt model fitting on deep learning, boosting and other algorithms in an attempt to develop an even more effective model.

9. Analyse the data using Inductive Decision Trees as an alternative means to determining feature importance.

10. Generate super-features based on the cluster heads of the existing features and use these new feature sets to fit new models, potentially generating a higher performing solution.

11. Re-run the analysis with the software application type as an additional feature in order to determine if the application type has any impact on software quality.

12. Analyse the SonarQube rule and metric severity values as additional features in order to determine if they have any predictive value for software defects.

13. Analyse the relationship between software application author and software defect outcomes and add this variable as an additional predictive feature.

It is anticipated that, as additional data is collected, more features are tested and other models are tested, the results found in this research will be bolstered, with the ultimate impact of improving the overall effectiveness of current and future software development programs.

References

[1] Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E., Turski, W.M.: Metrics and laws of software evolution - the nineties view. In: Proceedings of the 4th International Symposium on Software Metrics. METRICS '97, Washington, DC, USA, IEEE Computer Society (1997) 20–32

[2] Faris, T.: Safe and Sound Software: Creating an Efficient and Effective Quality System for Software Medical Device Organizations. ASQ Quality Press (2006)

[3] Campbell, G.A., Papapetrou, P.P.: SonarQube in Action. 1st edn. Manning Publications Co., Greenwich, CT, USA (2013)

[4] Fairley, R.E.: Tutorial: Static analysis and dynamic testing of computer software. Computer **11**(4) (April 1978) 14–23

[5] Bouwers, E., Visser, J., van Deursen, A.: Getting what you measure. Commun. ACM **55**(7) (July 2012) 54–59

[6] Chen, T.H., Shang, W., Nagappan, M., Hassan, A.E., Thomas, S.W.: Topic-based software defect explanation. Journal of Systems and Software **129** (2017) 79 – 106

[7] Tong, H., Liu, B., Wang, S.: Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning. Information and Software Technology **96** (2018) 94 – 111

[8] Mrinal Singh Rawat, S.K.D.: Software defect prediction models for quality improvement: A literature study. IJCSI International Journal of Computer Science Issues **9, Issue 5**(2) (Sep 2012)

[9] Brooks, Jr., F.P.: No silver bullet essence and accidents of software engineering. Computer **20**(4) (April 1987) 10–19

[10] Brooks, F.P.: The mythical man-month : essays on software engineering. Anniversary ed.. edn. Addison-Wesley, Reading, Mass. ; Don Mills, Ont. (1995)

[11] Malaiya, Y.K., Denton, J.: Estimating the number of residual defects [in software]. In: Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No.98EX231). (Nov 1998) 98–105

[12] Benington, H.D.: Production of large computer programs. Annals of the History of Computing **5**(4) (Oct 1983) 350–361

[13] Royce, W.: Managing the development of large software systems. In: Proceedings of IEEE WESCON. Volume 26 of WESCON 1970. (1970) 328–388

[14] Rumpe, B.: Modeling with UML Language, Concepts, Methods. Springer International Publishing : Imprint: Springer, Cham (2016)

[15] Hambling, B., van Goethem, P.: User Acceptance Testing: A Step-by-step Guide. BCS Learning & Development Limited (2013)

[16] Group, T.S.: Chaos report (1995) http://www.cs.nmt.edu/ cs328/reading/Standish.pdf.

[17] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mallor, S., Shwaber, K., Sutherland, J.: The agile manifesto. Technical report, The Agile Alliance (2001)

[18] Schwaber, K., Sutherland, J.: The scrum guide. Online, http://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf (nov 2017)

[19] Ono, T.: Toyota production system : beyond large-scale production. Productivity Press, Portland (1988)

[20] Beck, K.: Extreme programming explained : embrace change. 2nd ed.. edn. Addison-Wesley, Boston, MA (2005)

[21] Ahimbisibwe, A., Daellenbach, U., Cavana, R.Y.: Empirical comparison of traditional plan-based and agile methodologies: Critical success factors for outsourced software development projects from vendors perspective. Journal of Enterprise Information Management **30**(3) (2017) 400–453

[22] Stoica, M., Mircea, M., Ghilic-MICU, B.: Software development: Agile vs. traditional. Informatic economic **17**(4) (January 2013) 64–76

[23] Schwaber, K., Beedle, M.: Agile Software Development with Scrum. Agile Software Development. Prentice Hall (2002)

[24] Oberscheven, F.M.: Software quality assessment in an agile environment. Master's thesis, Radboud University Nijmegen (2013)

[25] August, T., Niculescu, M.: The influence of software process maturity and customer error reporting on software release and pricing. Management Science **59**(12) (December 2013) 2702–2726

[26] Haskins, B., Stecklein, J., Dick, B., Moroney, G., Lovell, R., Dabney, J.: 8.4.2 error cost escalation through the project life cycle. In: INCOSE International Symposium. Volume 14. (06 2004) 1723–1737

[27] Dawson, M., Burrell, D., Rahim, E., Brewster, S.: Integrating software assurance into the software development life cycle (sdlc). Journal of Information Systems Technology and Planning **3** (01 2010) 49–53

[28] Muhammad Dhiauddin Mohamed Suffian, S.I.: A prediction model for system testing defects using regression analysis. International Journal of Soft Computing And Software Engineering **2**(7) (Jul 2012)

[29] Marchenko A, A.P.: Predicting software defect density: A case study on automated static code analysis. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **4536 LNCS** (2007) 137–140

[30] Johnson, S.C.: Lint, a c program checker. In: COMP. SCI. TECH. REP. (1978) 78–1273

[31] Fenton, N.E., Neil, M.: A critique of software defect prediction models. IEEE Transactions on software engineering **25**(5) (1999) 675–689

[32] Rutar, N., Almazan, C.B., Foster, J.S.: A comparison of bug finding tools for java. In: 15th International Symposium on Software Reliability Engineering. (Nov 2004) 245–256

[33] Hinton, G., Sejnowski, T.: Unsupervised Learning: Foundations of Neural Computation. A Bradford Book. MCGRAW HILL BOOK Company (1999)

[34] Alpaydn, E.: Machine learning. Wiley Interdisciplinary Reviews: Computational Statistics **3**(3) (May 2011) 195–203

[35] Leottau, D.L., del Solar, J.R., Babuka, R.: Decentralized reinforcement learning of robot behaviors. Artificial Intelligence **256** (2018) 130 – 159

[36] Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics. Springer New York (2013)

[37] Kuhn, M.: Applied predictive modeling. Springer, New York, NY (2013)

[38] Rawlings, J., Pantula, S., Dickey, D.: Applied Regression Analysis: A Research Tool. Springer Texts in Statistics. Springer New York (2001)

[39] Loh, W.: Classification and regression trees. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery **1**(1) (January 2011) 14–23

[40] Hearst, M.A.: Support vector machines. IEEE Intelligent Systems **13**(4) (July 1998) 18–28

[41] Huang, Y.: Advances in artificial neural networks methodological development and application. Algorithms **2**(3) (August 2009) 973–1007

[42] Breiman, L.: Random forests. Machine Learning **45**(1) (Oct 2001) 5–32

[43] Kanungo, T., Mount, D.M., Netanyahu, N.S., Piatko, C.D., Silverman, R., Wu, A.Y.: An efficient k-means clustering algorithm: analysis and implementation. IEEE Transactions on Pattern Analysis and Machine Intelligence **24**(7) (Jul 2002) 881–892

[44] Everitt, B.S., Landau, S., Leese, M., Stahl, D.: Hierarchical clustering. Cluster Analysis, 5th Edition (2011) 71–110

[45] Kan, S.: Metrics and Models in Software Quality Engineering. Addison-Wesley (2003)

[46] Hartigan, J.A., Wong, M.A.: Algorithm AS 136: A K-Means clustering algorithm. Applied Statistics **28**(1) (1979) 100–108

[47] Thorndike, R.L.: Who belongs in the family? Psychometrika **18**(4) (Dec 1953) 267–276

[48] Kuhn, M.: The caret package (2017)

[49] Guyon, I., Weston, J., Barnhill, S., Vapnik, V.: Gene selection for cancer classification using support vector machines. Machine Learning **46**(1) (Jan 2002) 389–422

[50] Burez, J., Van den Poel, D.: Handling class imbalance in customer churn prediction. Expert Syst. Appl. **36**(3) (April 2009) 4626–4636

[51] Terry Therneau, B.A.: rpart: Recursive partitioning and regression trees (2018)

[52] Zorn, C.: A solution to separation in binary response models. Political Analysis **13**(2) (April 2005) 157–170

Glossary

ANOVA	Analysis of Variance.
AUC	Area Under the Curve.
C	Cost.
CP	Cost Complexity.
D	Decay.
LoC	Lines of Code.
M.SD	Mean Absolute Error Standard Deviation.
MAE	Mean Absolute Error.
MSLOCs	Millions of Source Lines of Code.
mtry	Minimum feature count used to TRY and grow a tree in a random forest.
R.SD	Root Mean Squared Error Standard Deviation.
RFE	Recursive Feature Elimination.
RMSE	Root Mean Squared Error.
ROC	Receiver Operator Characteristic.
RS	R-squared.
RSSD	R-squared Standard Deviation.
S	Size.
SQL	Structured Query Language.
SVM	Support Vector Machine.
SVN	Apache Subversion.

Appendices

A Supplementary Figures

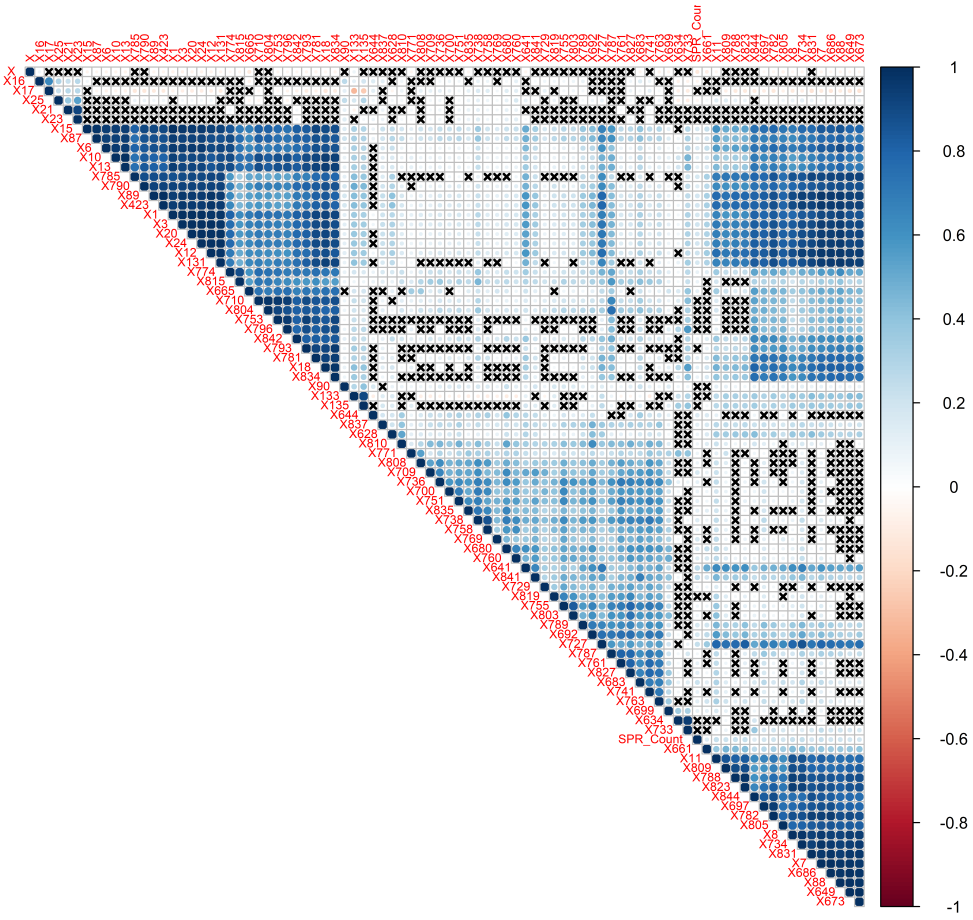

Figure 18: Correlation Plot of Features and Outcome

B Supplementary Tables

ID	Description
X1	Lines
X3	Lines of code
X4	Lines of code per language
X6	Classes
X7	Files
X8	Directories
X10	Functions
X11	Accessors
X12	Statements
X13	Public API
X15	Comment lines
X16	Comments (%)
X17	Public documented API (%)
X18	Public undocumented API
X20	Complexity
X21	Complexity /file
X23	Complexity /class
X24	Complexity in functions
X25	Complexity /function
X27	Functions distribution /complexity
X28	Files distribution /complexity
X87	Duplicated lines
X88	Duplicated blocks
X89	Duplicated files
X90	Duplicated lines (%)
X118	Directory cycles
X119	Directory tangle index
X120	File dependencies to cut
X121	Package dependencies to cut
X122	Directory edges weight
X131	Technical Debt
X132	Technical Debt on new code
X133	SQALE Rating
X134	SQALE Development Cost
X135	SQALE Technical Debt Ratio
X423	Duplicated blocks

ID	Description
X626	Classes should not be empty
X628	equals(Object obj) and hashCode() should be overridden in pairs
X631	Methods named equals should override Object.equals(Object)
X632	public static fields should always be constant
X634	Modifiers should be declared in the correct order
X637	Octal values should not be used
X640	object == null should be used instead of object.equals(null)
X641	Declarations should use Java collection interfaces such as List rather than specific implementation classes such as LinkedList
X644	Loggers should be private static final and should share a naming convention
X645	IP addresses should not be hardcoded
X647	The Object.finalize() method should never be overriden
X648	Loop counters should not be assigned to from within the loop body
X651	The default unnamed package should not be used
X653	Non-constructor methods should not have the same name as the enclosing class
X655	Execution of the Garbage Collector should be triggered only by the JVM
X660	equals(Object obj) should be overridden along with the compareTo(T obj) method
X661	Generic wildcard types should not be used in return parameters
X665	The members of an interface declaration or class should appear in a pre-defined order
X667	Constants should not be defined in interfaces
X671	super.finalize() should be called at the end of Object.finalize() implementations
X673	Class variable fields should not have public accessibility
X675	Short-circuit logic should be used in boolean contexts
X677	Identical expressions should not be used on both sides of a binary operator
X679	Classes from com.sun.* and sun.* packages should not be used
X680	String literals should not be duplicated
X681	Exception types should not be tested using instanceof in catch blocks
X682	java.lang.Error should not be extended

ID	Description
X683	Avoid commented-out lines of code
X687	Nested code blocks should not be used
X689	static final arrays should be private
X690	for loop incrementers should modify the variable being tested in the loop's stop condition
X692	Empty statements should be removed
X693	Values passed to SQL commands should be sanitized
X697	switch statements should have at least 3 cases
X699	System.out and System.err should not be used as loggers
X700	Math operands should be cast before assignment
X703	The Object.finalize() method should never be called
X704	Credentials should not be hard-coded
X705	Assignments should not be made from within sub-expressions
X708	Public methods should throw at most one checked exception
X709	Loops should not contain more than a single break or continue statement
X710	@Override annotation should be used on any method overriding (since Java 5) or implementing (since Java 6) another one
X713	ConcurrentLinkedQueue.size() should not be used
X714	Long suffix L should be upper case
X715	Deprecated elements should have both the annotation and the Javadoc tag
X720	Objects should not be created to be dropped immediately without being used
X723	Exceptions should not be thrown in finally blocks
X725	Loop conditions should be true at least once
X726	Exception classes should be immutable
X727	Exception handlers should preserve the original exception
X729	Empty arrays and collections should be returned instead of null
X731	Class names should comply with a naming convention
X732	Switch cases should end with an unconditional break statement
X733	Method names should comply with a naming convention
X734	Methods should not be too complex
X736	Switch cases should not have too many lines
X739	String.valueOf() should not be appended to a String

ID	Description
X740	Labels should not be used
X741	Collection.isEmpty() should be used to test for emptiness
X742	Case insensitive string comparisons should be made without intermediate upper or lower casing
X744	Primitive wrappers should not be instantiated only to perform a to String conversion
X746	Throwable and Error classes should not be caught
X747	super.clone() should be called when overriding Object.clone()
X751	Local variables should not shadow class fields
X753	Overriding methods should do more than simply call the same method in the super class
X754	Printf-style format strings should not lead to any runtime unexpected behavior
X755	Methods should not be empty
X756	Unused labels should be removed
X757	Object.wait(...) and Condition.await(...) should always be called inside a while loop
X758	Collapsible if statements should be merged
X760	Expressions should not be too complex
X761	Lambdas and anonymous classes should not have too many lines
X762	wait(...) should be used instead of Thread.sleep(...) when a lock is held
X763	Unused private fields should be removed
X764	Synchronisation should not be based on Strings or boxed primitives
X766	Cloneables should implement clone
X767	Conditions in related if/else if statements should not be duplicated
X769	Constants should be declared final static rather than merely final
X770	Non-static class initializers should not be used
X771	Unused method parameters should be removed
X772	The Array.equals(Object obj) method should never be used
X774	Avoid too complex class
X775	Parentheses should be removed from a single lambda input parameter when its type is inferred
X777	Object.finalize() should remain protected (versus public) when overriding

ID	Description
X779	instanceof operators that always return true should be removed
X781	Unused local variables should be removed
X782	Switch statements should end with a default case
X785	Floating point numbers should not be tested for equality
X787	Literal boolean values should not be used in condition expressions
X788	Variables should not be declared and then immediately returned or thrown
X789	Return of boolean expressions should not be wrapped into an if-then-else statement
X790	Useless parentheses around expressions should be removed to prevent any misunderstanding
X792	Redundant casts should not be used
X793	Useless imports should be removed
X795	switch statements should not have too many case clauses
X796	Utility classes should not have a public constructor
X801	Package declaration should match source file directory
X805	A field should not duplicate the name of its containing class
X808	Fields in a Serializable class should either be transient or serializable
X809	Methods should not have too many parameters
X810	Try-catch blocks should not be nested
X814	Return statements should not occur in finally blocks
X815	Nested blocks of code should not be left empty
X817	BigDecimal(double) should not be used
X819	Right curly braces should be located at the beginning of lines of code
X820	If statement conditions should not always evaluate to true or to false
X821	Collections should not be passed as arguments to their own methods
X822	IllegalMonitorStateException should never be caught
X823	Tabulation characters should not be used
X824	System.exit(...) and Runtime.getRuntime().exit(...) should not be called
X825	hashCode and toString should not be called on array instances
X827	Throwable.printStackTrace(...) should never be called

ID	Description
X830	Package names should comply with a naming convention
X831	Statements should be on separate lines
X832	Reflection should not be used to check non-runtime annotations
X834	Local variable and method parameter names should comply with a naming convention
X835	Strings literals should be placed on the left side when checking for equality
X836	Type parameter names should comply with a naming convention
X837	Deprecated code should be removed eventually
X838	Interface names should comply with a naming convention
X839	FIXME tags should be handled
X841	TODO tags should be handled
X842	Field names should comply with a naming convention
X844	Constant names should comply with a naming convention
X845	toString should not return null

Table 56: SonarQube Rule and Metric Reference Table