Masters in Computer Science
June 2023

Exploring Machine Learning Techniques and Metric Feature Pairs for Software Defects Prediction

Muhammad Ahmad Imran Rafique
Lokesh Kola

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Masters in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Author(s):
Muhammad Ahmad Imran Rafique
E-mail: murf21@student.bth.se

Lokesh Kola
E-mail: lokl20@student.bth.se

University advisor:
Shahrooz Abghari, PhD
Department of Computer Science

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Background :- Software Defect Prediction is important in identifying potential issues in software systems, improving software quality, and reducing development and maintenance costs. Diverse methodologies, ranging from rule-based systems to machine learning, offer various approaches to defect prediction. However, the challenge of class imbalance due to fewer instances of defective entities impacts model performance. This thesis investigates machine learning techniques and feature-metric pairs for defect prediction, including metric selection, using machine learning methods, and dataset creation using techniques like SMOTE to address class imbalance and see how they impact predictive performance.
Objectives :- The objective of this research is to i) explore feature-metric pairs ii) see what aspects of the software impact prediction quality iii) see how different machine learning algorithms behave and offer results iv) check the impact of class imbalance and oversampling techniques v) analyze and check how models behave and what can be done to improve them
Methods :- We created two datasets using SMOTE and one without SMOTE, applied various feature selection techniques, and scored them to find which were important, and used machine learning models to check how useful they are in predicting software defects. The performance metrics recorded are time, accuracy, F1 score, precision, and recall. A scikit-learn pipeline was created in order to handle these tasks from data addition to prediction.
Results :- Our results indicate that most size and documentation metrics are important features, despite their simplicity, in predicting defects. Despite the models not being tuned enough, ensemble techniques worked really well in the process. SMOTE trained datasets yielded better F1 Scores but most of their precision quality is not acceptable. Voting classifier models performed the most balanced in terms of accuracy,f1 score with the non-SMOTE version at 75%,57% and the SMOTE one at 75%,58% respectively.
Conclusions :- The conclusion shows that size and documentation metrics, despite their simplicity, are noticeably important factors for defect prediction and that ensemble techniques can be used and further tuned for the task as they offer the most balanced performance in the experiment.

Keywords: Class Imbalance, Feature Selection, Machine Learning, Software Metrics, Metric Feature Pairs, Software Defects

Acknowledgments

We would like to thank our supervisor Dr. Shahrooz Abghari for his kind and supportive supervision to make this thesis possible. Furthermore, we would like to thank our examiner Emilia Mendes for her critique and evaluation of our work. Most of all, we would like to thank our Family and Friends for their love and support.
- Ahmad and Lokesh

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software defects are defined as errors or unacceptable behaviors of the source code that lead to the software functioning in an incorrect or unexpected way [1]. Software defects may arise due to a plethora of reasons such as programming errors, design implementation errors, software environment malfunction, or attempting to parse inputs that were not anticipated beforehand, and no appropriate exception handling was put in place to deal with them. Often, in the maintenance phase of the Software Development Life Cycle (SDLC), writing a patch without catering to the exact initial development environment may cause another error once the software is put into production. These errors may also occur due to inadequate testing by virtue of test cases, technique employed, or lack of time and resources.

The errors can show up at any point of the SDLC from requirements gathering to implementation and testing. Timely handling of these defects is imperative as otherwise leads to increased development costs, delays, and decreased user base satisfaction. Moreover, they contribute to the buildup of technical debt.

Maintenance and implementation [2] are the most expensive stages of the SDLC where businesses try to reduce costs as much as possible. To this end, much work has been done in terms of formulating and improving appropriate methodologies pertaining to software testing and implementation, such as maintaining a separate quality assurance team, performing code reviews, acceptance testing, and automated testing, which are utilized to ensure reliable and correct software development. In modern software engineering, thanks to machine learning and deep learning approaches, there has been an increase in prediction-based testing strategies or helping aids [3].

Software Defect Prediction (SDP) is a collection of techniques used in software development, primarily in testing phases, to estimate the likelihood of a bug occurring. These techniques try to identify regions of the source code that are prone to defect so that appropriate corrective actions may be performed before the software is released. With the advent of sophisticated statistical methods, improved computational ability, and machine learning methods, these techniques have seen a massive improvement in the past few years.

Software Defect Prediction typically relies on historical data analysis of what is known as software metrics to build models that predict defect proneness of the software component. Their utility lies mainly in their speed in identifying problem zones and as an aid for developers and testing teams to allocate timely and appropriate resources to deal with the defect-prone zone.

SDP methods are typically divided into the following categories, namely:

1. Static analysis: Here, the information about the software is recorded without

ever executing the software. The information is recorded in the form of software metrics (measurements) that capture different attributes of the source code such as size and complexity. They can further be divided into categories like direct code (the metrics taken from source code) or indirect ones (the metrics taken from comments and documentation).

2. Rule-based analysis: Here, user-defined rules are applied over the software to check if any violation has been found. These rules are agnostic to the medium of programming when it comes to logic. However, programming language syntax requires a different set of rules to be used. These rules do not require software to be run either.

3. Dynamic analysis: Here, the defect information is pointed out during the execution of the software. Modern Integrated Development Environment (IDE) uses these where syntactical and logical errors are pointed out based on pre-defined rules. Some of these rules change depending on the programming language being employed but largely, the logical rules remain the same. Similar to rule-based, these methods are automated and know where the point of defect is and can often suggest corrective actions that can be taken based on predefined or user-defined rules.

4. Statistical analysis: Here, machine learning methods are applied to historical data, such as those collected from static analysis methods, to predict the future likelihood of a defect. These methods are highly adaptable to the data and offer quicker highlights into the trends of any topic under discussion and perform as a decision support aid.

Software defects can be handled by considering the following two approaches:

1. Reactive: This approach follows an event that has occurred that resulted in an error. User feedback is one way where developers can take that feedback into account and implement a patch if the error results from the software.

2. Proactive: This method employs rule-based or statistical methods for identifying defect-prone zones prior to their occurrence.

1.1 Scope of the thesis

The thesis project focuses on proactive approaches. Proactive methods enable one to make rules or methods based on prior or historical data to make decisions related to declaring a class defect prone. Within this, machine learning methods are particularly helpful in creating models that express the relationship between explanatory and response variables or even observing patterns when the data is not labeled, albeit the quality of the data being considered is the caveat. Functionally, it is also a faster approach than utilizing manual methods [1].

The data for software defect prediction primarily comes from static analysis tools which measure different attributes of the data. The Chidamber Kemerer Java Metrics (CKJM) [4] for instance are one famous software measurement tailored for object

oriented nature of Java. With the introduction of improved hardware and deep learning techniques, there has been a rise of using source code itself as the dataset (Using Abstract Syntax Tree Token (ASTT) to study semantic and structural information about the defect prone zones [5].

The field and research area of Software Defect Prediction (SDP) is an ever relevant and an interesting topic to explore because of the evolving software development methods and a focus on good programming practices in order to curb security related issues as well and even more so due to the shift towards prediction based software testing strategies [3] . The problems in the domain can be summarized in the following aspects:

1. Explaining the relationship between attributes and defect

   Researchers have used different attributes based on their findings to try to make a solution for software defect prediction. There is also an argument as to what level of metrics are suitable for use. Researchers have compiled metrics at the class or file level. With the availability of higher and better performing computers, statement level parsing has also been done. There is a lack of consensus as to what level metrics should be explored and that changes the attribute set to be considered for the problem between similar studies.

2. Lack of standardized evaluation criteria for performance assessment.

   The evaluation criteria to check the performance of solutions is different between the studies performed. Some have recommended precision and recall as the evaluators [6]. As most metrics datasets suffer a class imbalance problem precision has been recommended as an evaluation metric to must report [7] among others like F1 Score, variance scores, and Area under the ROC Curve (AUC). There are no basic criteria set and with different attributes and experimental setup used in different experiments, a comparison of multiple studies with any proposed solution becomes questionable.

3. Class Imbalance issue

   There are datasets such as NASA MDP and Promise [8] that are used as benchmark datasets for defect prediction tasks along with studies making their own datasets depending on what angle they wish to explore. In most cases, the dataset is imbalanced, usually in favor of the non-defective classes. As such, it becomes an arduous task while employing oversampling or undersampling techniques to build a balanced dataset to pass to models as their real-world utility at times becomes questionable due to the somewhat artificial nature of the data.

4. Lack of utility when it comes to performing cross project defect prediction.

   The metrics dataset collected about projects has an issue when used to predict defects in a different project with the same metrics. The increasing number of complex software systems gave rise to the idea of using a

prediction model trained on one dataset and used on another and while feasible, it has been found that development environment, methods, data distributions of target and source are different which makes using projects in a cross-project manner difficult to directly use [9].

5. Lack of a standard framework involving different datasets and methods

   Different researchers have used different techniques and datasets to train their defect prediction systems, meaning there is a lack of standardized procedures or frameworks in the field, which is understandable given that most solutions cater to a specific scale of software and have issues implemented in a cross-project manner owing to lack of robust techniques and comparison methods [10], difference of outputs in repeated runs [11], and small consideration towards addressing security related aspects in software defect predictions, makes this an arduous task.

6. Economic viability has rarely been discussed.

   The proposed solutions have rarely discussed their economic viability of themselves. Given time and resources, a robust model may be constructed but discussing whether the defect prediction should even be used and whether it is useful. Bathia et al. [12] study for instance attempted to create a cost evaluation framework to assess the viability of a defect prediction system. Their results showed that a range of 20-40 % of defects are viable to use, and certain algorithms perform better given the number of faults the dataset has. However, improved hardware capability and optimized algorithms in modern times have rendered some of these concerns useless and in the end, the quality of the data and measurements calculated are the larger contributors to ascertaining the economic viability of a defect prediction solution.

This thesis concerns the exploration of viable software metrics from a larger dataset and models that contribute best towards explaining a software defect. As such, we will:

1. Explore software metrics. As elaborated in Section 3, this thesis will explore a large software metric dataset of 80 metrics collected from small to medium scale software.

2. Extract informative metrics using data mining techniques. Feature selection techniques will be applied, coupled with a machine learning model to find which metric model pair performs the best.

3. Observe and discuss the performance of the metric model pair.

4. Compare how SMOTE [1]generated and manually balanced datasets impact training and validation.

5. Analyze and explain the results produced.

[1]Synthetic Minority Oversampling Technique - used to generate rows for minority class labels

6. Create a pipeline script that performs extraction to prediction in one go for ease of use.
7. A conclusion of the results, note down some takeaways and notes for how to work on this topic in the future.

The thesis does not include:

1. Extracting features using any tool from any project as an existing project dataset is in use.
2. A tool to graphically explain the results of predictions from the pipeline.
3. A study to see the impact of pipeline script or thesis findings on real users.
4. Performance of models in a real-world software environment.

1.2 Background

1.2.1 Machine Learning

Machine Learning (ML) is a subset of Artificial Intelligence. ML deals with the systems, which gain knowledge by extracting patterns from the data. ML is of two types

1. **Supervised ML:** The Supervised ML deals with the data, which has a target variable. These models are used to predict the target variable by using the other features in the data.
2. **Unsupervised ML:** This Unsupervised ML deals with unlabelled data. These models are used for clustering the data on the basis of their similarity.

1.2.2 ML Algorithms

Support Vector Machine(SVM)

This algorithm is used in supervised prediction (labeled data) for regression and classification problems. SVM identifies the hyper-plane in between different types of data. The algorithm works by finding the optimal hyperplane that separates the data into different classes or predicts the target variable. In real world, data is usually not linearly separable which is why SVM have a feature called a "kernel trick" which allows data to be transformed into a higher dimension in order to find and establish the decision boundary.

Extreme Gradient Boost (XGBoost)

An open source implementation of the gradient boosting algorithm combines many weak learners to explain a target variable. As a combination of gradient based and tree based models, it creates series of models that aim to correct the error found in the last one (bagging approach) and have shown to improve predictive accuracy and model interpretability.

k-Nearest Neighbour(kNN)

The kNN algorithm is a simplified ML algorithm, which is used for classification problems. This algorithm classifies the data points based on their neighbor's classification. The model performance is based on the choosing right k value. By hyperparameter tuning chosen k value likely provides high model performance.

Multi-Layer Perceptron(MLP)

This is a neural network consisting of multiple interconnected perceptrons working in a feedforward network manner (information flows in one direction). The neurons in each layer receive input from the previous, apply an activation function to weighted sum of inputs and pass to next layer and uses backpropogation to adjust the weights of the learner. They are computationally expensive and may overfit on data and have a difficulty when it comes to interpreting the model.

Naive Bayes (NB)

This algorithm requires few tunable parameters and is usable in both discrete and continuous value prediction. It is fast and by design scalable and a popular choice as a baseline model. Based on Bayes theorem, it is a statistical method that describes the probability of an event based on prior knowledge of conditions pertaining to the event. It assumes feature independence meaning that presence and absence of a feature has no impact on presence and absence of another feature.

Logistic Regression(LR)

The LR algorithm is useful in predicting the probability of belonging to a particular class of binary categories. This algorithm may also be used as a potential baseline performer. It works by modeling the probability that an instance belongs a particular class given its features by learning a set of weights (also called coefficients) that describe the relationship between the features and the log-odds of belonging to the particular class.

Random Forest(RF)

A simplistic algorithm that uses an ensemble of decision trees and a combination of their results to extract the final output. It works by making a subset of trees and selects best features to split the data into further trees. The results are aggregated to provide a final prediction.

1.2.3 Evaluation Metrics

The evaluation metrics, which has been used for measuring the performance of the model.

Accuracy

The accuracy of the model will be based on the predicted values. The predicted values are checked with true values. The higher accuracy of the model leads to better model performance.

$$Accuracy(A) = \frac{TP + TN}{TP + FP + TN + FN} \tag{1.1}$$

Precison

This evaluation method measures the correctly predicted observations over the total predictions made. This is the accuracy of the positive predictions made. In layman's terms, precision is a measurement of quality where a higher precision indicates, that more relevant results are being returned than irrelevant ones.

$$Precision(P) = \frac{TP}{TP + FP} \tag{1.2}$$

Recall

This evaluation method measures the relevancy of the predicted observation. It measures the completeness of positive predictions. In layman's terms, recall is a measurement of quantity where a higher recall indicates, that most of the relevant results are being returned.

$$Recall(R) = \frac{TP}{TP + FN} \tag{1.3}$$

F1 Score

This is the harmonic mean of precision and recall and is used to explain the quality of observations made. Loosely, this can further explain how correct the accuracy of the model is. As it puts equal emphasis on both precision and recall, the evaluation metric may be biased towards one class.

$$F1Score = \frac{2 \times Recall \times Precision}{Recall + Precision} \tag{1.4}$$

1.3 Defining research questions

RQ1: Which metric set is most useful in explaining a relation to a software defect?

> **RQ1.1**: Which ML technique is the most efficient in software defects prediction?

Motivation : *RQ1 and its sub-question will allow us to explore and identify the metrics, that are more useful in predicting software defects and the models, that are further better at predicting them. Metrics like F1 Score, precision, recall, etc. are to be utilized as evaluators. In this way, this project will compare against models used previously in a similar domain or between themselves and also highlight potential future areas of work.*

RQ2: How does the dataset creation using SMOTE or manually impact the prediction of the models?

Motivation : *Plenty of literature, some cited above, used over-sampling or under-sampling methods to make representative data but that data is still somewhat, artificial, albeit a popular technique in literature. Our motive is to test how the quality of the models is affected when the data used is SMOTE generated or by manual representative sampling techniques.*

Chapter 2

Related Work

2.0.1 Search Criteria

For the purpose of our thesis work, we used Google Scholar, IEEExplore, ACM Digital Library, and Researchgate to get the largest bulk of related work. Despite Google Scholar collecting a broad variety of papers, it was limiting in getting to the exact results, therefore, other commonly used libraries with peer-reviewed works were chosen.

Search Strings: **("All Metadata": software defect prediction) AND ("All Metadata": Machine Learning) AND ("All Metadata": Class Imbalance) OR ("All Metadata": Deep Learning) OR ("All Metadata": Data Mining) AND ("All Metadata": Feature Selection)**

The language of the papers searched were in English only. With our keywords set, we set criteria to view works from 2016 and later.

Search Strings, language, and time make up our initial inclusion/exclusion criteria

Table 2.1: Related Search Criteria

Step No.	**Action**	**Paper**
1	Initial inclusion/exclusion criteria	48 342
2	After Tools Used	189
3	Papers for initial References	50
4	Papers after some important out of year added	55

With results in the order of thousands, 48 342, In Step 2, we utilized all-MiniLM-L6-v2 sentence transformer model [13] at hugging face[1] where we focused on how relevant the abstracts of the papers were in order to further narrow them down according to our scope, And then used a search tool ConnectedPapers[2] to see what related works a relevant document was cited with. These tools were used as assists while we read through the literature and attempted to find relevant work. With careful consideration, in step 3, the papers came down to 50, which included papers relevant to understanding the context of our research area.

During the course of the thesis, some papers were identified that were older than the year-wise filter or important to add as important context for any assertions made in the thesis which brought related works and references used in the thesis, up to 55.

[1]https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

[2]https://www.connectedpapers.com

With the limitation of language, year filter, and time available for research, it is possible that some relevant literature was missed as a comprehensive systematic literature review was not employed as a research method for our thesis needs, however, we used the literature at hand to identify gaps and/or interesting researchable topics and continued with the required work.

2.0.2 Relevant Works

Ran Li et al. [14] analyzed ensemble methods on NASA's Metrics Data Program (MDP) dataset, a dataset intended for defect prediction research, where they found Random Forest (RF) to be a good performing model. In their experiment, they employed the Synthetic Minority over-sampling Technique (SMOTE) to generate a balanced dataset, as the defect datasets generally consist of either fewer defects or more defects compared to non-defects.

Du et al. [15] in their study discuss a hybrid technique where the majority class (class with the most frequent data label) is under-sampled via logistic regression to fix miss classification rates while minority classes are over-sampled via a combination of k-means with SMOTE to generate more artificial data. Afterward, the new datasets are put in bagging and a voting ensemble (of logistic regression, support vector, and decision tree(DT)), where they are compared for their classification strengths. The novel thing here was the effort in defect prediction dataset creation to make it all fair.

Anh et al. [16], in their study, introduced a hybrid method of prediction by combining feature reduction and a deep learning model to make their defect prediction model. Using Kernel PCA, they reduced the features or metrics down to the most useful ones, and then a deep neural network was used to study the non-linear relationship of the data. To address the class imbalance, they did not use a sampling technique but a weighted technique which made this an interesting study to consider.

There have also been attempts at trying to connect all possible entities that impact software creation and triangulate that information to detect defects. Other than conventional software metrics such as line of code, complexity, and CKJM suite. Yihao Li et al. [17] proposed a tri-relation model where they added a number of developers and the number of commits made as part of their metrics dataset.

They wanted to observe how the developer impacts the process alongside creating a means to localize fault-prone areas in what they call a tri-relation network. The metrics data is collected from six open-source projects. Their experiment design aims to check whether the tri-relation network model will predict fault proneness and indicate bug possibility. Their results show that the tri-relation network outperforms existing ones they compared to, but they only used six open-source projects in the experiment. The experiment is also difficult to reproduce due to a lack of a standardized issue tracker system.

Khuat et al. [18], in their evaluation study of software defect prediction, used an ensemble of Logistic regression, Support vector machine, multi-layer perceptron, Naive Bayes, DT, and k nearest neighbor to capture the non-linear relationship of the metrics data while addressing the class imbalance of the defect set as well. Their evaluation study focused more on sampling aspects and how they impact the training and whether they were needed or not. To this end, they developed two ensemble

schemas and used random under-sampling, over-sampling, and SMOTE on them to check for significant changes. The random under-sampling showed significant differences compared to the SMOTE-based ensembles, albeit not the other combinations.

Aside from using metrics, using the Abstract Syntax Tree (AST) of the source code and converting them into tokens for processing is another approach, which Deng et al. [19] have done. The authors collected source code data at the file level from various Java projects and converted them to a token format they fed to their Long Short Term Memory (LSTM) model. The objective was to traverse the AST tree and understand the semantic and contextual information of the code to map to a defect. Compared to other state-of-the-art models, the proposed method performed better but not so significantly. However, using the source code file itself rather than the metrics, which represent features of code in numbers, is an interesting approach as well to try proactively dealing with defects.

Jian Li et al. [20] used the same AST token principle to train a defect prediction model via a Convolutional neural network (CNN). To handle the class imbalance in their data, they duplicated the minority class enough times till it matched the majority class in the count. CNN is employed to capture structural information of the source code and coupled with handcrafted traditional metrics, their approach shows an improvement over previous similar studies. However, the generalizability is low as 7 Java-based projects were used only.

Azeem et al. [21] performed a literature review to understand how ML approaches for bug detection or code smell as they are used, are impacted. They observed that the developer's subjectiveness towards bug detection, the definition of bug collected by any tool used, lack of agreement between how different detectors work, and creating a threshold for defect proneness, make a size fits all approach towards defect prediction difficult. Their study consisted of over 2000 papers and assessed ones that used proper ML approaches. Their study mainly provides a general overview of how algorithms, independent variables, and training strategy impact code smell detection however their approach is yet to be tested on a benchmark dataset to observe in action.

Mattias et al. [22] in their study explored how source code file metrics could be used in conjunction with respective test metrics to better detect faults. Their focus was on source code data at a file level. They utilized metrics such as the CKJM, Line of code, McCabe complexity, and Software testing and reliability early warning metric suite or STREW as test metrics to relate to defects. From that, they created and collected a common dataset to be used in ML models.

To minimize model creation complexity, they eliminated features based on a filtering approach called Correlation-based Feature Selection (CFS) by Hall [23] and best first search to further evaluate CFS. Their study showed that a combination of software and test metrics together has no profound impact on defect prediction as compared to when the model was trained simply on software metrics, but the potential to train the model on test metrics to find source code defects exists, as they demonstrated.

The interesting thing about this study was their approach to incorporating other factors that directly or indirectly impact defects in source code. The study used few metrics given their experiment design and resources but the potential to incorporate more measurements to understand the relation of software metrics to defects exists.

Jacob et al. [24] in their study utilized a voting ensemble made of RF, AdaBoost, and naive Bayes to address the task of fault prediction. They tested a wrapper-based vs heuristic-based approach feature selection method and showed that the wrapper-based based performed generally better among all datasets, averaging an 86% F1 Score. They used the NASA MDP dataset with 22 metrics, of which five are different lines of code measures, four are based on Halstead measures, three are McCabe metrics, eight are derived from Halstead measures, and one is a branch count and a categorical attribute. The feature selection approaches made this an interesting paper to consider.

They employed a wrapper and heuristic feature elimination strategy each to get 10 metrics from each, with 50% of the metrics being the same in both. They compared their voting ensemble against RF, DT, support vector, multi-layer perceptron, AdaBoost, etc., and showed that their implementation had better performance once evaluated. The dataset and feature set they used were somewhat small even before reduction and results on a larger diverse dataset against a deep learning method would do an interesting experiment in terms of performance and time.

The argument of which evaluation metrics are useful to use has also been a concerning point in the context of Software defect prediction which Morcasa et al. [25] also tried to study. They figured that classifying faulty modules using evaluation functions like ROC, AUC, etc is highly dependent upon on threshold used for the scoring function. They argue that ROC is better than some other metrics to define defect proneness and that evaluation should be restricted to a threshold zone with good performance.

They use a new metric called the Ratio of Relevant areas to overcome problems in existing ones and show that this metric works well along with other metrics and combinations of them. This study was conducted over a real-life dataset from the SEACRAFT repository. This metric is customizable for software engineering teams but costs to use is a caveat along with the fact that dataset change can change the Region of interest and impact results.

Eman et al. [26] approached the problem by finding models that fit a developer's perspective on software quality improvement tagged as refactoring. Their study used a corpus of design related refactoring activities from 3795 open source Java projects and assessed their structural metrics and anti-pattern enhancement changes. This approach took in a developer's perspective on software quality and showed that not all state-of-the-art metrics are useful in making the relationship between internal quality and end product. This study was useful because as an extension, we can see how developer-related information is useful as software measurements tend to implicitly embed certain developer processes however environmental factors do impact software development.

Varma et al. [27] discuss the challenges that surround the software defect prediction domain and how they were and are being tackled from a historical to a current point of view. The lack of diverse data at a commercial level is one such problem that results in diverse methods to approach the problem in context. Software complexity is increasing but the lack of availability of large-scale datasets for this express purpose renders these solutions tailor-made and have reduced utility in a cross-project manner. Another problem is the feature set to use.

The lack of data makes finding meaningful features difficult and decreases the

usage in a cross-project manner. They comment that making SDP a part of the software development lifecycle will help reduce costs in activities like maintenance, along with making robust Software Quality Assurance (SQA) activities. For that, SDP methods need to be formalized with their process creation, model, and evaluation criteria. The paper is also useful for anyone trying to build a SDP model and what steps from data creation to model selection should be considered.

Shantini et al. [28] in their study aim to check the performance of an ensemble approach for a Support Vector Machine in fault prediction. The stress towards usage of software metrics for software quality aspects. They assert that an ensemble approach is better than most classical defect prediction models and they performed the experiments on the Eclipse Package level and NASA KC1 dataset (class-level metrics). Their evaluation, in terms of RMSE, AUC-ROC showed a better fault prediction classification rate than individual approaches. This paper was useful in understanding how the granularity of metrics i-e class level or package level, impacts the prediction quality along with seeing the effectiveness of ensemble methods.

Nasir uddin et al. [29] in their study utilized boosting algorithms on NASA MDP datasets to build their SDP model. Their approach aims to evaluate the performance of the gradient-boosting algorithms LightGBM, XgBoost, and CatBoost. Their findings also show how search-based techniques and model tuning impact results. Their results found GridSearchCV based LightGBM outperformed other considered models and ones used in similar studies, in terms of accuracy, precision, and recall. The consideration in this study focuses on better models and procedures for better prediction, over data handling itself. Search-based techniques are one way for feature selection but parameter selection is an exhaustive and time-consuming method, albeit, a valid consideration in these studies.

Tessema et al. [30] in their study focus on a different angle of viewing defects, via code change requests for a just-in-time prediction. They seek to use the code itself and see how changes to code impact its proneness to defect. They augmented publically available datasets with six change request-based metrics to test their considered models. They used boosting, ensemble, neural network, and probabilistic models.

The metrics they use are Time to fix, Number of developers, priority, severity, Number of comments, and Depth of discussion, and validated their approach on data collected from Eclipse, Mozilla, and Bugzilla issues tracking. 19 out of 20 times, their approach showed improved performance over existing just-in-time approaches, and F1 Score improved by an average of 4.8%, 3.4%, 1.7%, 1.1% and 1.1% using models like AdaBoost,XgBoost, Deep Neural Net, Random Forest, Logistic regression respectively. This study was interesting due to the data used in the approach. Even implicitly considering external entities like developers as a feature was interesting along with other features work. From an ethical standpoint, direct developer information is not used but maybe with numeric metrics, such change based metrics in tandem might produce interesting results.

Kakkar et al. [31] in their study aimed to develop a framework for attribute selection in SDP models using ensemble, lazy learners and search based feature selectors. They used the 5 NASA MDP dataset CM1,JM1,KC1,KC3,PC1, C/C++ language-based dataset, to test their approach with 38,22,22,40,38 features in them, reduced to 6,7,7,4,9 respectively. The feature selector used an exhaustive search method, best first and greedy, further ranked using a Chi-Square test and correlation tests. The

t statistic on the results showed that there was no difference in accuracy before and after feature selection, which also means that a similar result can be obtained by using much smaller feature sets and saving time.

Liu et al. [32] in their paper explore deep learning approaches towards detecting code smells or defects. As manual methods are tedious, automatic or semi-automatic methods are proposed in modern literature and practice. Simple statistical models also have their limitations but a deep learning approach automatically selects the best or most useful features and is able to build the complex mapping of features to label (defective or not).

Their research used well-known code smells of feature envy, long method, large class, and misplaced class. The training data was created by introducing refactoring that drops the software quality however that in itself was the challenge, as in the domain, to have large enough training data to use. Their approach generated the labeled training data without any human intervention and their results have shown significant improvement over the state of the art. The interesting aspect, other than the deep learning approach, is the kind of data used for defect detection. Loosely speaking, it moves from traditional numeric metrics to source code aspects which similar studies employing deep learning are putting a larger focus on.

Chapter 3

Method

A formal experiment will be performed to investigate **RQ1**, by utilizing feature reduction/selection techniques in order to extract the best metrics/features which influence the defect prediction. **RQ1.1** is the continuity of the experiment performed as part of **RQ1**, where the reduced feature set will be used to train different ML algorithms, and their performance will be compared against each other. **RQ2** is an analysis of the quality of the data used in the experiment to observe the SMOTE-based balancing technique and manual representative balancing technique, albeit not balanced (that is the labels do not represent a perfect 50% split for each label), impacts on the training process. The experiment setup is as follows:

1. Performing data preprocessing techniques on the raw PROMISE dataset.
2. Creating two datasets using random selections, can address class imbalance however, one dataset is constructed using real dataset values while the other is using under/over-sampling to address the class imbalance. This way, the impact on validation will also be observed to determine whether there is any difference in performance and how.
3. Performing the feature engineering on both datasets by utilizing appropriate feature selection techniques and evaluating their impact on predictions.
4. Applying selected ML methods on the datasets with tuned hyperparameters.
5. Performing the stratified 10-fold cross-validation on both datasets and evaluating the performance of the models.
6. Identifying the most efficient ML method to predict defects by using evaluation metrics like accuracy, precision, recall, and F1 Score.

3.1 Dataset

The metrics data for this project is the PROMISE dataset which comes from the Unified Bug dataset version 1.2 by Rudolph et al. [33]. The Unified bug dataset is a collection of multiple Java-based metrics datasets that have been used in prior studies and the authors have collected all of them into a single repository for ease of research. The PROMISE dataset is among this public repository which consists of several Apache projects with classical metrics data of 20 metrics (add citation and example studies) and an additional 60 modern metrics data, which compiles up to 80

total metrics across multiple versions of the projects in the dataset. The PROMISE dataset is frequently used in related research works [34]. Other characteristics that made this dataset a choice are as follows:

1. The dataset holds information about the projects across multiple versions, which makes exploring from a within-project point of view possible.

2. The dataset projects are of varying sizes and defect densities (point to a table) which makes exploring from a cross-project point of view possible.

3. The dataset is labeled and can easily be modified to suit specific research needs based on how the problem is approached.

4. The dataset is imbalanced and thus allows us to address the class imbalance context and impact of balancing techniques explorable.

5. The dataset implicitly embeds a good developer practice of documenting or commenting on their work, at the class level and makes it interesting to explore how they relate to a defect, without explicitly adding any developer-related data (citation needed on why good, mainly ethical)

6. The data is compiled at a class level and as such, it offers the perfect granular view of the software while allowing us to address the reusability of metrics across versions of the project.

7. As per literature class level [28] or even file level [35] metrics have shown good utility but the problem is dependent on context. As modern programming styles utilize object-oriented patterns, The data compiled at a class level offers a better look into the relationship between modules as well as how they may impact defect prediction, furthermore allowing us to understand qualitative information such as cohesion, coupling, and complexity in software development.

Table 3.1 shows the datasets inside the Unified Bug Dataset PROMISE repository along with its class count and Thousand lines of code (kLOC) and percentage of classes with bugs.

Table 3.2 and 3.3 show the metrics and their category with respect to object-oriented styles used in this thesis. In both tables, the features specified (63) are the initial metric set we will use after removing redundant or unneeded features. More details in (supplement info)

Table 3.1: PROMISE dataset Projects

Project	Number of Classes	Percentage of Buggy Classes	kLOC
Ant 1.3	125	16.00	33
Ant 1.4	178	22.47	43
Ant 1.5	293	10.92	72
Ant 1.6	351	26.21	98
Ant 1.7	745	22.28	116
Camel 1.0	339	3.83	26
Camel 1.2	590	36.61	47
Camel 1.4	841	17.24	76
Camel 1.6	928	20.26	99
Ckjm 1.8	9	55.56	8
Forrest 0.6	6	16.67	19
Forrest 0.7	29	17.24	4
Forrest 0.8	32	6.25	3
Ivy 1.4	241	6.64	31
Ivy 2.0	352	11.36	54
JEdit 3.2	272	33.09	55
JEdit 4.0	306	24.51	63
JEdit 4.1	312	25.32	72
JEdit 4.2	367	13.08	88
JEdit 4.3	492	2.24	109
Log4J 1.0	135	25.19	10
Log4J 1.1	109	33.94	14
Log4J 1.2	205	92.20	23
Lucene 2.0	194	46.91	68
Lucene 2.2	246	58.54	111
Lucene 2.4	340	59.71	126
Pbeans 1	26	76.92	3
Pbeans 2	51	19.61	6
Poi 1.5	237	59.49	63
Poi 2.0	314	11.78	82
Poi 2.5	385	64.42	94
Poi 3.0	442	63.57	140
Synapse 1.0	157	10.19	20
Synapse 1.1	222	27.03	33
Synapse 1.2	256	33.59	46
Velocity 1.4	196	75.00	26
Velocity 1.5	213	66.20	33
Velocity 1.6	228	34.21	37
Xalan 2.4	723	15.21	104
Xalan 2.5	803	48.19	126
Xalan 2.6	885	46.44	154
Xalan 2.7	909	98.79	160
Xerces 1.2	440	16.14	65
Xerces 1.3	453	15.23	69
Xerces 1.4	547	72.39	74

*Bugs/kLOC is bugs per thousand lines of code. This qualitative aspect shows bug existence scaled with the size of the source code.

Table 3.2: Category of Metrics Used: Cohesion, Complexity and Coupling

Category	Abbreviation	Metric Name
Cohesion	LCOM5	Lack of Cohesion in Methods 5
	cam	Cohesion among methods of class
	cbm	coupling between methods
Complexity	NL	Nesting Level
	NLE	Nesting Level If Else
	wmc	weighted method per class
	max_cc	maximum McCabe cyclomatic complexity
	avg_cc	average McCabe cyclomatic complexity
	mfa	measure of functional abstraction
	dam	data access metric
Coupling	CBOI	Coupling between object classes
	NII	Number of incoming invocations
	NOI	Number of outgoing invocations
	cbo	coupling between objects
	rfc	response for class
	ca	Afferent coupling
	ce	Efferent Coupling
	moa	Measure of aggregation
Documentation	AD	API Documentation
	CD	Comment Density
	CLOC	Comment Lines of Code
	DLOC	Documentation Line of Code
	PDA	Public Documented API
	PUA	Public Undocumented API
	TCD	Total Comment Density
	TCLOC	Total Comment Line of Code

Table 3.3: Category of Metrics Used: Inheritance and Size

Inheritance	DIT	Depth of Inheritance Tree
	NOA	Number of Ancestors
	NOD	Number of Descendants
	NOP	Number of Parents
	ic	inheritance coupling
	noc	Number of Children
Size	LLOC	Logical Lines of Code
	NA	Number of attributes
	NG	Number of getters
	NLA	Number of local attributes
	NLG	Number of local getters
	NLM	Number of local methods
	NLPA	Number of local public attributes
	NLPM	Number of local public methods
	NLS	Number of local setters
	NM	Number of methods
	NOS	Number of statements
	NPA	Number of public attributes
	NS	Number of setters
	TLLOC	Total logical lines of code
	TLOC	total lines of code
	TNA	total number of attributes
	TNG	total number of getters
	TNLA	Total Number of Local Attributes
	TNLG	Total Number of Local Getters
	TNLM	Total Number of Local Methods
	TNLPA	Total Number of Local Public Attributes
	TNLPM	Total Number of Local Public Methods
	TNLS	Total Number of Local Setters
	TNM	Total Number of Methods
	TNOS	Total Number of Statements
	TNPA	Total Number of Public Attributes
	TNPM	Total Number of Public Methods
	TNS	Total Number of Setters
	npm	number of public methods
	loc	line of code
	amc	average method complexity

3.2 Dataset processing

As per Table 3.1, the projects inside the dataset are imbalanced in favor of either defect or non-defect. This has a potential for bias in the tasks and as such, efforts to minimize this bias as much as possible are undertaken. To this end, efforts have been made to create a balanced or representative dataset.

Since this project aims to check the impact of SMOTE generated data as well, 2 datasets will be made from the original pool of 45 datasets from the PROMISE dataset, one quasi-randomly balanced and the other randomly selected and balanced using SMOTE. In the latter, given the information we have about all datasets, there is a high probability that defective data labels will be fewer in number and will be subjected to oversampling. The SMOTE variant used is SMOTE Tomek [36] which basically tries to oversample the minority class and any collision with the majority class is removed. From a certain pool of SMOTE variants, this was deemed appropriate to use (more explained in section 4).

The following explains how the datasets were created:

1. **Dataset 1 - non-SMOTE**: The datasets are ranked first in ascending order based on size. Next, the percentage of defects percentage is ranked in ascending order. The two ranks are averaged and thus a balance of size and defect rate is created. The average rank is sorted in ascending order. The average value of the average rank is computed and datasets closest to that number are selected from the center and datasets from boundary values are selected. Afterward, datasets are randomly selected from either extreme to the average value. In the end, we get 25 projects combined into a dataset of 9 634 rows with 4 052 defective labels or 42% defective data and 5 582 non-defective labels.

2. **Dataset 2 - SMOTE balanced**: The datasets are ranked first in ascending order based on size. Next, the percentage of defects percentage is ranked in ascending order. The two ranks are averaged and thus a balance of size and defect rate is created. The average rank is sorted in ascending order and 5 columns are appended with random values between 0 and 1 generated between them using the average number as a seed value. The average of the random values is taken by row and the average column is then sorted in ascending order. The average of that column value is computed and all datasets above that value are selected. In the end, 24 projects are combined into a dataset of 9 992 rows with 3 757 defective labels or 37% defective data and 6 235 non-defective labels. SMOTE up-sampling is applied to the dataset after which we get a total of 11 664 rows with 5 832 labels for defective and non-defective each.

Considerations about creating the datasets were made to reduce bias and cherry-picking from the datasets. Additionally, the datasets have varying characteristics to what constitutes or explains a defect which is interesting as lots of information to use exists. Lastly, these characteristics in the collective data have the potential to be used in a cross-project manner (datasets of multiple distributions and sizes exist so can either be joined [37] or training on one project and prediction on a project of similar size are done [38]).

3.3 Dataset Characteristics

The two datasets created, while having a difference in creation method, have the same metrics, data, and characteristics.

1. The datasets have high cardinality. There are a number of unique values for each feature,
2. The datasets have high dimensions (initial 80, started to work with 64 explained later)
3. The datasets are largely nonlinear. Few features among the initial dataset have weak to mild relation between themselves while some have a strong relationship with the target variable, but none are strongly correlated with each other.
4. The datasets have features as numeric (software metric measurements are all numeric) and response as binary categorical (true/false)
5. In both datasets, there is multicollinearity in the data, albeit defined as weak per a rule of thumb.

3.4 Metric Selection and data preprocessing

The two datasets have some redundant metrics (multiple references of McCabe complexity and line of code metrics and CKJM metrics) and metrics pertaining to clone code (code snippet copied as is within a class and used elsewhere in another class or a local method), which is not relevant to our study and are thus removed. Thus, from a total of 80 metrics, we are reduced to 64 metrics, see Tables 3.2 and 3.3.

The models used in this study use the entire dataset to train and record accuracy subjected to a 10 Fold Cross Validation method, time, and F1 Score (more explained in the later section) as a basis for elaborating how feature selection impacts future training and in what way. Additionally, it will be a good way to compare how different feature selection algorithms impact training.

Given the characteristics of the data and using help from prior studies in the related works the following strategies have been selected.

1. (initial) All dataset features
2. (MutInfo) Mutual Information and selecting k best
3. (Fisher) Fisher Score
4. (PermImp) Permutation importance
5. (VIFthresh)Remove features based on threshold via Variance Inflation Factor
6. (RFErfc) Recursive Feature Elimination based on tree-based model

Other considerations included and reasons for not including are as follows.

1. Principal component Analysis but it is not a viable method for high cardinality non-linear data and so is its Kernel PCA variant. (Expand on it further)

2. Auto encoders are one such option but for the scope of this project and the dataset used, it is a time consuming and not optimal method to employ. (Expand on it further)

3. Pearson Correlation was considered but the high cardinality nature of data renders its usage improper.

The dataset having high cardinality is scaled between 0 and 1 for all purposes.

3.5 Model Selection

With features selected, the purpose of this thesis is to identify or test the ML methods that best elaborate the relationship between feature set and defect. The ML models should ideally

1. Scale with more data

2. Work in higher dimensions

3. Show utility in past literature of a similar nature.

The feature model combination is used to compare and check which version best explains the relationship between feature and defect Software defect prediction is a binary classification task and models that have been used in prior research fall into the following family of algorithms:

1. Probabilistic models: These models learn from data distributions to generate the likelihood of an entity belonging to a particular group. Examples include Naive Bayes and logistic regression. Used in research like [39].

2. Ensemble models: These models stack multiple learners in order to generate better predictions. Examples include RF, DT, and voting classifiers. Used in research like [28].

3. Distance based: These models work by computing the distance of unknown or new data to existing exemplar data to classify or predict their group. Examples include k nearest neighbor and Support vectors [28] [40].

4. Gradient Boosting: These models are a variation of ensemble techniques where multiple weak learners are combined in hopes of getting better performance. Examples include XGBoost. Used in research like [29].

5. Neural Networks: These models try to map complex information about the features of the data by themselves without little human intervention and make decisions on their own. Examples include Artificial neural networks. Used in research like [32].

In addition to the existing methods, a custom voting ensemble classifier will be built to check its performance against existing methods, using the top-3 performing algorithms. A voting classifier is used because of its noise mitigation effect and as per literature is a known performance booster [41]. The voting classifier will also have a different feature selection strategy. It is expected that chosen models for this ensemble will use different features on their own and as such, an intersection of features and a fallback option of union of features will be used in case there is no common feature set. Goutam et al. [42] used a ranked intersection and union technique, albeit for a sentiment analysis task. In our case, since the best model and feature pair are used, a ranking between them a second time is not needed.

A custom pipeline will also be constructed at the end to allow model selection and data entry all the way to model creation, prediction, and validation as well in order to speed up work.

Based on prior research and related works, the following models have been selected and tuned for use (along with the short hand form used in tables).

1. Gaussian Naïve bayes (gnb)
2. Logistic Regression (logr)
3. Random Forest Classification (rfc)
4. Xtreme Gradient Boosting (xgb)
5. Support Vector Classification (svm)
6. Multi-layer perceptron (mlp)
7. k Neighbors Classifier ($k$nn)

An additional model based on a voting ensemble with hard voting will be created using the top-3 best performing models in terms of F1 Score and time. Since each individual model may have a specific feature set it performs best on, for the voting classifier, we will adopt an intersection with union fallback to create the dataset. The accuracy drop is a valid consideration and consequence of such a method but is also the fairest construction considered.

Each algorithm is subjected to a Randomized Search Cross Validation for the purpose of hyperparameter tuning.

3.6 Training Method

The models will be run through multiple rounds of stratified 10-fold cross-validation and information about its evaluation metrics will be recorded into a data table and saved as a Comma Separated Value (CSV) file. Evaluation Metrics The following evaluation metrics will be recorded while running the stratified cross-validation on the models.

1. Accuracy

2. Time execution

3. F1 Score, precision, recall

F1 Score and Time execution will be useful in forming the voting ensemble. Precision and recall information related to the best performing model will also be reported.

3.7 Pipeline construction

To swiftly perform tasks pertaining to this thesis, Jupyter Notebooks were used and to make it easier for all tasks to be done swiftly, a Python-based pipeline function was constructed. This pipeline function

1. Connects various Python modules like scikit learn, numpy, and pandas

2. Takes as input the data, scaling function, feature selector strategy, and model

3. Performs cross validation inside and report accuracy, F1 Score, and time

To add to the pipeline function

1. Inbuilt validation data handler

2. Inbuilt model reports

3. Process an array of models and associated data

4. Make a full process report including what features were selected and removed along with reasoning and a full classification report.

5. A comment based on data like precision, recall, F1 Score, etc to give a one line decision about displayed output (for the model or array of models).

3.8 Validity and reliability of your approach

The approach is

1. Used considerations of design based on previous literature

2. Reduces the amount of bias in any feature selection method

3. The results can be compared amongst the same dataset research so generalizability is questionable.

4. The approach uses numeric software measurements. There are other approaches such as commit history, defects between last changes, or alternative code smell labels. At the very basic, it is expected the same data is used.

5. While a wide array of features and models are used, the basic datasets used in the projects belong to software that uses an object-oriented style which represents modern development standards and the expectation of new test models to also be object-oriented in nature.

6. Since only English language research papers were chosen, relevant research may have been left out.

3.9 Challenges Faced

In the course of the thesis work, there were some roadblocks which were mainly due to time constraints and handling the data. The thesis explored various feature selection methods, however, each method, used in various literature works, used diverse reasoning for their use and feature cut-off points. Other than literature, we had to find inspiration in practical work to decide certain considerations. Additionally, for our thesis, making a data set that fairly compared SMOTE and non-SMOTE versions was a challenge but thanks to our supervisor's input, that problem was solved. Lastly, again due to time and limiting scope, a proper deep-learning model was not implemented for the thesis however, as part of future works, deep-learning models are considered.

Chapter 4

Results

4.1 Feature from the non-SMOTE Data

The feature selection algorithms were run over the data manually built and tested with a combination of the models. The following results were obtained

1. Mutual Information: 'NOI', 'AD', 'CD', 'CLOC', 'DLOC', 'PDA', 'TCD', 'TCLOC', 'LLOC', 'NA', 'NLA', 'NM', 'NOS', 'NPA', 'TLLOC', 'TLOC', 'TNA', 'TNLA', 'TNOS', 'TNPA', 'rfc', 'loc', 'dam', 'mfa', 'cam', 'amc'

2. Fisher Score: 'TNLS', 'LLOC', 'DIT', 'ce', 'cbo', 'NOI', 'NPA', 'loc', 'LCOM5', CBOI', 'NII','avg_cc', 'dam', 'NL', 'cam', 'ca', 'max_cc', 'moa','CD', 'TNLPM', 'NM', 'NG', 'NLA', 'NOA', 'TNLA'

3. Permutation Importance: Each model exhibited a specific feature set so that is used.

   (a) Gaussian NB:'NG', 'NOA', 'cbm', 'TNM', 'NM', 'PDA', 'DIT', 'TNPM'

   (b) Logistic Regression: 'cbm', 'NOA', 'DIT', 'CLOC', 'amc', 'NOI', 'TNA', 'TCLOC', 'TLLOC'

   (c) Random Forest: Eliminated all ergo all could be used

   (d) XGBoost: 'TLOC', 'CLOC'

   (e) Support Vector : 'CD', 'cbm', 'mfa', 'dam', 'TCD', 'CLOC', 'cam', 'ic', 'amc', 'AD', 'TNPM', 'NOI', 'DLOC', 'TCLOC', 'TLOC', 'NOP', 'NLG', 'ce', 'TNM', 'NA', 'loc', 'LLOC', 'TNLG'

   (f) Multilayer Perceptron: 'cbm', 'NOI', 'NOA', 'amc', 'DIT', 'CLOC', 'ic', 'NS', 'TNS', 'NM', 'dam', 'TNPM', 'moa'

   (g) KNN: 'mfa', 'AD', 'dam', 'TCD', 'DIT', 'cam', 'CD', 'NOA', 'CLOC', 'DLOC', 'TCLOC', 'NL', 'TNPM', 'PDA', 'TNLPM', 'NLPM', 'TLOC', 'TNLM', 'cbo', 'max_cc', 'LLOC', 'NLM', 'wmc', 'NII', 'TLLOC', 'NOD', 'noc'

4. Variance Inflation Factor: 'LCOM5', 'NL', 'NLE', 'CBOI', 'NII', 'AD', 'NOD', 'NOP', 'noc', 'dam', 'moa', 'mfa' 'cam', 'ic', 'cbm', 'max_cc', 'avg_cc'

5. Recursive Feature Elimination: 'NL', 'CBOI', 'NII', 'NOI', 'AD', 'CD', 'CLOC', 'DLOC', 'PDA', 'PUA', 'TCD', 'TCLOC', 'DIT', 'NOA', 'NOD', 'LLOC', 'NA',

'NG', 'NLA', 'NLG', 'NLM', 'NLPM', 'NM', 'NOS', 'NPA', 'NS', 'TLLOC', 'TLOC', 'TNA', 'TNG', 'TNLA', 'TNLG', 'TNLM', 'TNLPM', 'TNM', 'TNOS', 'TNPA', 'TNPM', 'TNS', 'wmc', 'cbo', 'rfc', 'ca', 'ce', 'npm', 'loc', 'dam', 'mfa', 'cam', 'cbm', 'amc', 'max_cc', 'avg_cc'

4.2 Feature from the Smote Data

The feature selection algorithms were run over the data subjected to SMOTE Tomek, built and tested with a combination of the models. Depending on the feature elimination strategy chosen, the final dataset length changed.

1. Mutual Information: 'NOI', 'AD', 'CD', 'CLOC', 'DLOC', 'PDA', 'TCD', 'TCLOC', 'LLOC', 'NA','NG', 'NM', 'NOS', 'NPA', 'TLLOC', 'TLOC', 'TNA', 'TNOS', 'TNPA', 'loc','mfa', 'cbm', 'amc'

2. Fisher Score: 'TNS', 'NM', 'NLPA', 'dam', 'npm', 'DLOC', 'NLG', 'rfc', 'LCOM5', 'CBOI', 'NOI', 'amc', 'cbo', 'NL', 'ic', 'ce', 'cam', 'max_cc', 'NII','NPA', 'NG', 'NS', 'TLOC', 'NOP', 'TNM'

3. Permutation Importance: Each model exhibited a specific feature set so that is used.

   (a) Gaussian NB: Eliminated all ergo all could be used

   (b) Logistic Regression: 'NOA', 'cbm', 'mfa', 'DIT', 'ic', 'NM', 'NS', 'NOP', 'TCD', 'NG', 'AD', 'CLOC', 'NOI', 'TNS', 'NLE', 'amc', 'LCOM5', 'wmc', 'TNLM', 'PUA'

   (c) Random Forest: 'dam', 'TLOC', 'AD', 'DLOC', 'TNPA', 'NPA', 'cbo', 'TNLM', 'NLA', 'DIT', 'ce', 'NLE', 'NOI', 'NL'

   (d) XGBoost : 'AD', 'TLOC', 'PUA', 'dam', 'CLOC', 'rfc', 'mfa', 'cbm', 'cam', 'NPA', 'PDA', 'CBOI', 'NOI', 'amc', 'TNPM', 'TLLOC', 'ce', 'NA', 'LLOC', 'ic', 'TCD', 'DLOC', 'ca', 'TNM', 'loc', 'TNS', 'TCLOC', 'NOS', 'TNLPM', 'TNLA', 'NLE'

   (e) Support Vector: 'AD', 'dam', 'ic', 'DIT', 'NOI', 'CLOC', 'cbm', 'DLOC', 'TNLPM', 'TCLOC', 'NLPM', 'NOD'

   (f) Multilayer Perceptron: 'NOA', 'cbm', 'ic', 'mfa', 'NM', 'NS', 'amc', 'DIT', 'NOI', 'TNS', 'CLOC', 'NG', 'NOP', 'TCLOC', 'TNPM', 'TNLPM', 'NLG', 'TNG', 'NL', 'AD', 'TNLG', 'dam', 'TNLS'

   (g) KNN : 'dam', 'AD', 'mfa', 'cam', 'DIT', 'CD', 'TCD', 'NOA', 'ic', 'NM', 'NLE', 'cbm', 'TNA', 'NA', 'TNPA', 'NPA', 'avg_cc', 'NOD'

4. Variance Inflation Factor: 'LCOM5', 'NL', 'NLE', 'CBOI', 'NII', 'AD', 'NOD', 'NOP', 'noc', 'dam', 'moa', 'mfa', 'cam', 'ic', 'cbm', 'max_cc', 'avg_cc'

5. Recursive Feature Elimination: 'CBOI', 'NII', 'NOI', 'AD', 'CD', 'CLOC', 'DLOC', 'PDA', 'TCD', 'TCLOC', 'DIT', 'NOA', 'LLOC', 'NA', 'NG', 'NLA', 'NLM', 'NLPM', 'NM', 'NOS', 'NPA', 'NS', 'TLLOC', 'TLOC', 'TNA', 'TNG',

'TNLA', 'TNLG', 'TNLM', 'TNLPM', 'TNM', 'TNOS', 'TNPA', 'TNPM', 'TNS', 'wmc', 'cbo', 'rfc', 'ca', 'ce', 'npm', 'loc', 'dam', 'mfa', 'cam', 'cbm', 'amc', 'max_cc', 'avg_cc'

4.3 Data Obtained

4.3.1 Accuracy

Table 4.1 and 4.2 shows the accuracy of the model and features when subjected to both smote and non-SMOTE. In both tables, we see a nearly similar accuracy despite different sampling techniques used however Bagging and Boosting algorithms like Randomforest (rfc) and XgBoost (xgb) appeared to have improved accuracy with SMOTE.

Table 4.1: Mean Accuracy (%) of 10 cross fold (Non-Smote)

features	gnb	logr	rfc	xgb	svc	mlp	knn
initial	0.6322	0.6312	0.6175	0.6089	0.4988	0.6394	0.6121
MutInfo	0.6299	0.6255	0.6064	0.6102	0.5154	0.6330	0.6131
Fisher	0.6306	0.6247	0.6158	0.6123	0.4966	0.6237	0.6121
PermImp	0.6106	0.6346	0.6146	0.6287	0.5645	0.6490	0.6129
VIFthresh	0.6174	0.6283	0.6140	0.6235	0.5744	0.6283	0.5949
RFErfc	0.6309	0.6277	0.6132	0.6021	0.5230	0.6396	0.6126

Figure 4.1: Mean Accuracy (%) of 10 cross fold (Non-Smote)

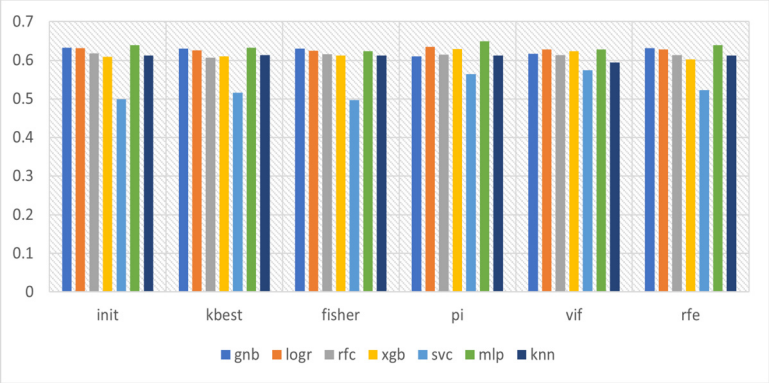
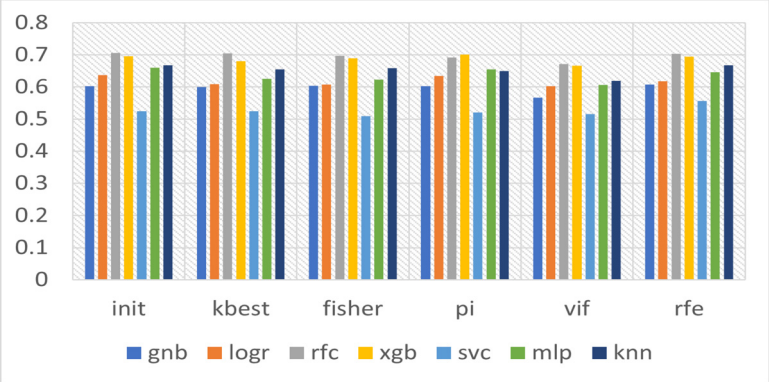


Table 4.2: Mean Accuracy (%) of 10 cross fold (Smote)

features	gnb	logr	rfc	xgb	svc	mlp	knn
initial	0.6023	0.6368	0.7059	0.6960	0.5243	0.6599	0.6682
MutInfo	0.6004	0.6092	0.7045	0.6803	0.5243	0.6253	0.6543
Fisher	0.6038	0.6078	0.6974	0.6895	0.5088	0.6224	0.6583
PermImp	0.6023	0.6347	0.6921	0.7014	0.5203	0.6545	0.6495
VIFthresh	0.5663	0.6029	0.6714	0.6664	0.5159	0.6060	0.6186
RFErfc	0.6081	0.6171	0.7037	0.6941	0.5567	0.6453	0.6674

Figure 4.2: Mean Accuracy (%) of 10 cross fold (Smote)



4.3.2 F1 Score

Table 4.3 and 4.4 show the Fscore of the models and features when subjected to both smote and non-smote. Fscore will be our voting classifier decision criteria. In the non-SMOTE arrangement, FScore is not so high and good but in SMOTE arrangement, we see boosting and bagging algorithms especially exhibit a good Fscore.

Table 4.3: Mean F1 Score (%) of 10 cross fold (Non-Smote)

features	gnb	logr	rfc	xgb	svc	mlp	knn
initial	0.4042	0.4104	0.5130	0.5023	0.5725	0.4931	0.4912
MutInfo	0.3418	0.4049	0.5182	0.5090	0.5557	0.4568	0.4975
Fisher	0.3984	0.3944	0.5110	0.5058	0.5424	0.4064	0.4919
PermImp	0.3361	0.3860	0.5072	0.5146	0.5695	0.4249	0.4963
VIFthresh	0.3527	0.4263	0.4909	0.4897	0.5823	0.4362	0.4697
RFErfc	0.4069	0.3987	0.5089	0.4889	0.5590	0.4939	0.4896

Figure 4.3: Mean F1 Score (%) of 10 cross fold (Non-Smote)

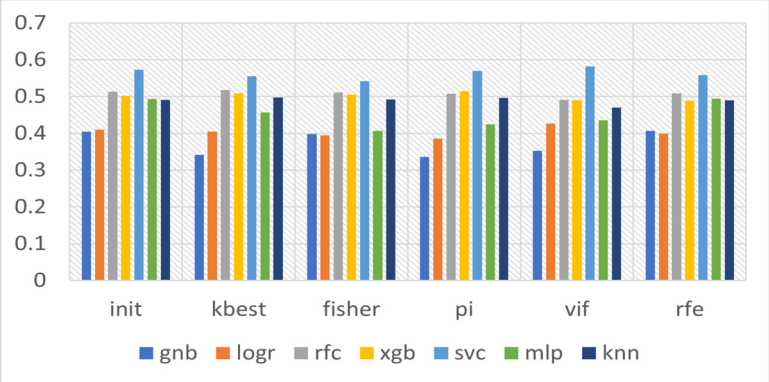
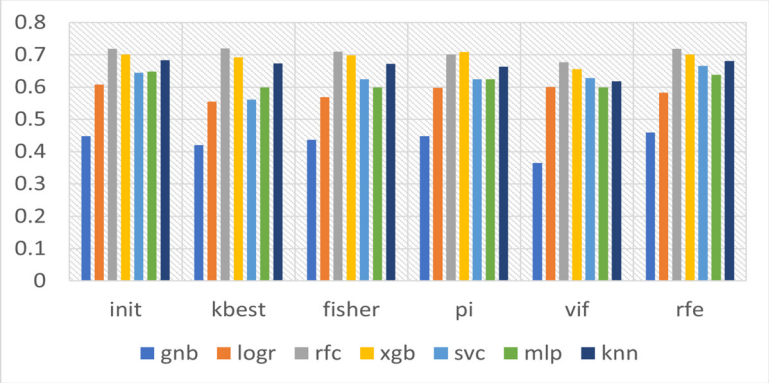


Table 4.4: mean F1 Score (%) of 10 cross fold (Smote)

features	gnb	logr	rfc	xgb	svc	mlp	knn
initial	0.4483	0.6081	0.7190	0.7011	0.6439	0.6481	0.6837
MutInfo	0.4199	0.5544	0.7196	0.6918	0.5614	0.5997	0.6728
Fisher	0.4371	0.5694	0.7097	0.6981	0.6248	0.5995	0.6727
PermImp	0.4483	0.5983	0.7011	0.7081	0.6242	0.6242	0.6627
VIFthresh	0.3653	0.6004	0.6766	0.6554	0.6275	0.5985	0.6177
RFErfc	0.4595	0.5833	0.7185	0.7015	0.6656	0.6383	0.6813

Figure 4.4: Mean F1 Score (%) of 10 cross fold (Smote)



4.3.3 Time execution

Table 4.4 and 4.5 shows the execution time of the model and features when subjected to both smote and non-SMOTE. Feature size and algorithmic complexity impact the time.

4.3.4 Status of the Voting classifier

The top-3 models using Dataset 1 are

1. Random Forest with Mutual Information based features
2. XGBoost with permutation importance features

Table 4.5: mean Execution Time (seconds) of 10 cross fold (Non-Smote)

features	gnb	logr	rfc	xgb	svc	mlp	knn
initial	0.0077	0.0705	2.2938	0.4635	1.6115	15.6489	0.0018
MutInfo	0.0032	0.0616	1.9844	0.2894	1.0893	12.3173	0.0011
Fisher	0.0074	0.0532	1.6178	0.2612	1.1804	10.5981	0.0011
PermImp	0.0016	0.0168	2.0802	0.1183	1.1321	5.9721	0.0013
VIFthresh	0.0026	0.0446	1.0751	0.2420	1.1141	5.8657	0.0009
RFErfc	0.0073	0.0921	2.4679	0.4458	1.4173	14.0470	0.0018

Table 4.6: mean Execution Time (seconds) of 10 cross fold (Smote)

features	gnb	logr	rfc	xgb	svc	mlp	knn
initial	0.0093	0.1182	2.4831	0.5542	2.0453	17.6355	0.0021
MutInfo	0.0037	0.0362	2.0704	0.3506	1.3268	14.5865	0.0012
Fisher	0.0054	0.0653	2.1648	0.3523	1.4078	14.1193	0.0015
PermImp	0.0089	0.0578	1.1858	0.3711	1.3087	12.6673	0.0012
VIFthresh	0.0033	0.0425	1.4728	0.2555	1.5453	6.2045	0.0010
RFErfc	0.0080	0.0792	2.8312	0.4802	1.6375	15.2786	0.0019

3. Support vector with Variance Inflation Factor

The feature set is 'NLA', 'TLOC', 'NOI', 'TNA', 'dam', 'TCLOC', 'TNPA', 'CD','CLOC', 'DLOC', 'TLLOC', 'AD', 'cam', 'PDA', 'NPA', 'TNOS', 'loc','NOS', 'amc', 'rfc', 'NM', 'TNLA', 'LLOC', 'TCD', 'NA', 'mfa'.

The accuracy of this arrangement is 67% with an F1 Score of 52% when run through a stratified 10-fold cross-validation model continuously.

The top-3 models using Dataset 2 are

1. Random Forest with Mutual Information based features

2. XGBoost with permutation importance features

3. kNN with all initial features

The feature set is 'DLOC', 'mfa', 'NOI', 'TCD', 'NOS', 'CLOC', 'LLOC', 'amc', 'TCLOC','TLOC', 'AD', 'NA', 'PDA', 'cbm', 'loc', 'NPA','TLLOC'

The accuracy of this arrangement is 66% with an F1 Score of 67% when run through a stratified 10-fold cross-validation model continuously.

4.3.5 Performance with blind dataset

The model arrangements are subjected to a test against an unseen dataset which contains data, largely not seen by either dataset arrangement.

This dataset borrows project files from the Unified Bug Dataset PROMISE projects that were not used in either of the two datasets above was used with 5 078 rows and a 30% defect rate.

Table 4.7: The best model-feature sets (No. of features in brackets)

Non -Smote feature arrangement	**Smote feature arrangement**
gnb with RFErfc (53)	gnb with RFErfc (49)
logr with VIFThresh (18)	logr with initial(56)
rfc with MutInfo (26)	rfc with MutInfo(23)
xgb with permImp (2)	xgb with permImp (31)
svc with VIFThresh (18)	svc with RFErfc(49)
mlp with RFErfc (53)	mlp with initial(63)
knn with MutInfo (26)	knn with initial(63)
Voting(RFC+XgBoost+SVM) (26)	Voting(RFC+XgBoost+kNN) (17)

To illustrate performance, Table 4.7 shows model-feature arrangements that yielded good Fscores, along with the new voting classifiers from each data arrangement are used to validate.

Table 4.8 and 4.9 show the performance of the models-feature arrangements with precision and recall as well. In work, precision greater than 50% is a good base to say that the models are working correctly at least more than half the time. From 4.8, only the voting classifier is acting in a balanced manner with good accuracy and FScore. In 4.9, random forest, multilayer perceptron, and the voting classifier are working correctly more than half the time. Although recall is a factor to look into, if our precision is bad, then it becomes meaningless to work on the model further.

Table 4.8: non-SMOTE model feature validation results summarized

Pair	**accuracy (%)**	**precision (%)**	**recall (%)**	**F1 Score (%)**
gnb RFErfc	29	29	100	45
logr VIFThresh	30	29	100	45
rfc MutInfo	39	32	96	48
xgb permImp	34	31	99	47
svc VIFThresh	70	22	1	3
mlp RFErfc	32	27	79	41
knn MutInfo	64	19	7	11
voting	75	57	57	57

	precision	recall	f1-score	support
False	0.00	0.00	0.00	3591
True	0.29	1.00	0.45	1487
accuracy			0.29	5078
macro avg	0.15	0.50	0.23	5078
weighted avg	0.09	0.29	0.13	5078

(a) gnb with RFErfc

	precision	recall	f1-score	support
False	0.97	0.01	0.02	3591
True	0.29	1.00	0.45	1487
accuracy			0.30	5078
macro avg	0.63	0.50	0.24	5078
weighted avg	0.77	0.30	0.14	5078

(b) logr VIFThresh

	precision	recall	f1-score	support
False	0.91	0.15	0.26	3591
True	0.32	0.96	0.48	1487
accuracy			0.39	5078
macro avg	0.61	0.56	0.37	5078
weighted avg	0.73	0.39	0.33	5078

(c) rfc MutInfo

	precision	recall	f1-score	support
False	0.93	0.08	0.14	3591
True	0.31	0.99	0.47	1487
accuracy			0.34	5078
macro avg	0.62	0.53	0.31	5078
weighted avg	0.75	0.34	0.24	5078

(d) xgb permImp

	precision	recall	f1-score	support
False	0.71	0.98	0.82	3591
True	0.22	0.01	0.03	1487
accuracy			0.70	5078
macro avg	0.46	0.50	0.42	5078
weighted avg	0.56	0.70	0.59	5078

(e) svc VIFThresh

	precision	recall	f1-score	support
False	0.60	0.13	0.21	3591
True	0.27	0.79	0.41	1487
accuracy			0.32	5078
macro avg	0.44	0.46	0.31	5078
weighted avg	0.50	0.32	0.27	5078

(f) mlp with RFErfc

	precision	recall	f1-score	support
False	0.70	0.88	0.78	3591
True	0.19	0.07	0.11	1487
accuracy			0.64	5078
macro avg	0.44	0.47	0.44	5078
weighted avg	0.55	0.64	0.58	5078

(g) knn MutInfo

	precision	recall	f1-score	support
False	0.82	0.83	0.82	3591
True	0.57	0.57	0.57	1487
accuracy			0.75	5078
macro avg	0.70	0.70	0.70	5078
weighted avg	0.75	0.75	0.75	5078

(h) voting

Figure 4.5: Detailed output of classification report in validation of non-SMOTE based models

	precision	recall	f1-score	support
False	0.72	0.87	0.79	3591
True	0.39	0.20	0.26	1487
accuracy			0.67	5078
macro avg	0.56	0.54	0.53	5078
weighted avg	0.63	0.67	0.64	5078

(a) gnb with RFErfc

	precision	recall	f1-score	support
False	0.80	0.67	0.73	3591
True	0.43	0.60	0.50	1487
accuracy			0.65	5078
macro avg	0.62	0.64	0.62	5078
weighted avg	0.69	0.65	0.66	5078

(b) logr VIFThresh

	precision	recall	f1-score	support
False	0.84	0.79	0.81	3591
True	0.55	0.63	0.58	1487
accuracy			0.74	5078
macro avg	0.69	0.71	0.70	5078
weighted avg	0.75	0.74	0.74	5078

(c) rfc MutInfo

	precision	recall	f1-score	support
False	0.83	0.71	0.77	3591
True	0.48	0.64	0.55	1487
accuracy			0.69	5078
macro avg	0.65	0.68	0.66	5078
weighted avg	0.72	0.69	0.70	5078

(d) xgb permImp

	precision	recall	f1-score	support
False	0.96	0.63	0.76	3591
True	0.51	0.93	0.66	1487
accuracy			0.72	5078
macro avg	0.73	0.78	0.71	5078
weighted avg	0.83	0.72	0.73	5078

(e) svc VIFThresh

	precision	recall	f1-score	support
False	0.81	0.79	0.80	3591
True	0.52	0.56	0.54	1487
accuracy			0.72	5078
macro avg	0.67	0.68	0.67	5078
weighted avg	0.73	0.72	0.73	5078

(f) mlp with RFErfc

	precision	recall	f1-score	support
False	0.84	0.72	0.77	3591
True	0.49	0.66	0.57	1487
accuracy			0.70	5078
macro avg	0.67	0.69	0.67	5078
weighted avg	0.74	0.70	0.71	5078

(g) knn MutInfo

	precision	recall	f1-score	support
False	0.84	0.77	0.80	3591
True	0.54	0.64	0.58	1487
accuracy			0.73	5078
macro avg	0.69	0.71	0.69	5078
weighted avg	0.75	0.73	0.74	5078

(h) voting

Figure 4.6: Detailed output of classification report in the validation of Smote-based models

Table 4.9: Smote model feature validation results

Pair	accuracy (%)	precision (%)	recall (%)	F1 Score (%)
gnb RFErfc	67	39	20	26
logr initial	65	43	60	50
rfc MutINFO	74	55	63	58
xgb permIMp	69	48	64	55
svc RFErfc	72	51	93	66
mlp initial	72	52	56	54
knn initial	70	49	66	57
voting	73	54	64	58

4.4 Comparison against other studies

In such machine learning experiments, a comparison is often not doable in a fair manner as the exact experiment environment has to be replicated in order to test models with the data or parameters. That said, for the sake of putting performances in perspective, we will compare our best models against the best models of similar studies where numeric metrics were used, the same dataset or its subsets were used and defect prediction was the goal. The performance comparison is made using F1 Score. Few deep learning models are also used to see how useful they are against simple tuned machine learning models.

Table 4.10: Smote model feature validation results

Model	Type	F1 Score (%)
SSDL for CSDP [43]	ML	43
CPDP-OSS [44]	ML	51
Voting Classifier non-SMOTE	ML	57
Voting Classifier Smote	ML	58
Random Forest with MutualInfo Features	ML	58
Ensemble with Random undersampling [18]	ML	65
Voting Classifier with Wrapper Approach [24]	ML	65
MK-TCNN [45]	DL	49
BSLDP [46]	DL	66
DP-Transformer [47]	DL	71

ML = Machine Learning, DL= Deep Learning, Models in **bold** are the ones created in this study.

Deep learning models have generally performed better against machine learning ones which can be attributed to design and or approach. However, in machine learning approaches, we see that ensemble techniques yield good results too. Ensembles are good as they improve the predictive performance against a single classifier [48]. Our models are not tuned or optimized as we compare them but they still show good base scores, which may further be improved, as is the case with ensemble techniques. Our method to improve our models are to focus on other feature selection methods and parameter tuning, as advised by Faseeha et al [49].

Chapter 5

Analysis and Discussion

Tables 4.1-4.2 show accuracy, 4.3-4.4 F1 Score, and 4.5-4.6 time in seconds for both SMOTE and non-SMOTE datasets. While the accuracies of the two approaches are a mixed bag, the F1 Score of SMOTE-based dataset models are overall quite better than the non-SMOTE version.

Recall, that an oversampling strategy was employed on the Smote-based dataset where we used the Tomek link variant that eliminates collisions (rows with the same feature values but different labels). In essence, we have reduced some majority class labels that could have been important in understanding the state of non-defect. To put them all into numbers, considering the feature sets used in table 4.8, the training data changed as follows from the initial 9 992 rows, with 6 235 non-defects, 3 757 defects to

1. MutInfo - 11 728 rows with 5 864 instances each (371 collisions)
2. RFErfc - 11 662 rows with 5 831 instances each (404 collisions)
3. permImp - 11 636 rows with 5 818 instances each (417 collisions)
4. tailored for voting - 11 718 rows with 5 859 instances each (376 collisions)

These collisions were not removed in the non-SMOTE version but comparing results from Table 4.8 and 4.9, non-SMOTE F1 Scores are quite near their SMOTE-based counterparts albeit SMOTE-based ones are higher overall. Studying the intricacies of these collisions in detail was beyond the scope of this thesis however it is an interesting problem to consider.

Considering tables 4.1, 4.3, and 4.8, we do see low accuracy and high F1 Score, a consequence of imbalanced datasets. Figure 4.1 (a) for instance shows that gnb RFErfc version precision and accuracy are almost equal with a high F1 Score, meaning that performance on the minority class is best for that model, that it has learned about minority features better than the majority. The reverse phenomena can be seen for combinations like svc VIFThresh in Figure 4.1 (e) which has a high accuracy but dismal F1 Score with degraded performance when it comes to classifying defective classes. mlp RFErfc Figure 4.1 (f) has a good accuracy, F1 Score, and recall ratio but it still tends towards one class over the other. The voting classifier ensemble in Figure 4.1 (h) albeit has a lower F1 Score than accuracy and still shows the best balanced results among the no SMOTE version as 4.1 (a-g) internally shows low precision (lower than 0.5) for minority classes indicating the high false positive occurrences for minority labels (defective) whereas Voting ensemble version is the

only one with over 0.5 precision and recall and is balanced in identifying defective classes.

Considering Table 4.2, 4.4, and 4.9, the accuracy is higher than the F1 Score in all best versions indicated in Table 4.9, almost every version shows higher recall than precision for defective labels. The F1 Score in Table 4.9 does indicate better validation results than the versions in Table 4.8, however, figure 4.2 illustrates the reason why. High recall than precision shows that relevant results are being displayed more often but results in many false positives, however as a rule, precision also has to be greater than 0.5. In Figure 4.2 (a),(d),(g), the results are dismal showing degraded model performance in validation.

Figures 4.2(f) and 4.2(h) are balanced versions in the sense that internally, more positive labels are being pointed out more often and at a good accuracy. In summary, SMOTE version despite having a good F1 Score does have a high false positive rate just like the non-SMOTE version, however, it has improved accuracy and F1 Score and is able to identify defective labels more often.

As such, it can be deduced that the model and feature selection methods require more tuning and that a better searching algorithm for features may be used.

The quality of data surely has the larger impact on the algorithm but what is interesting to note is that speaking strictly on the validation data experiment, SMOTE-generated data did not have such a huge impact over the non-SMOTE data, and in fact, the potential collision on similar rows different label was left but exploring that aspect is beyond the scope of this thesis.

5.1 Response for RQ1

When considering every model and feature combination, almost all the features were used once. Following are the categories and groups of metrics that were used most prominently independent of model and data sampling strategy.

Table 5.1: Prominent Features Selected

Category of Metrics	**Feature selected**
Size	loc,LLOC,NA,NM,NOS,NPA,TLLOC, TLOC,TNA,TNOS,TNPA,amc
Documentation	AD,CD,CLOC,DLOC,PDA,TCD,TCLOC
Coupling	NOI,cam,dam,rfc
Cohesion	cbm
Complexity	mfa

From our experiments, Table 5.1 presents a combination of the most useful sets of features that can be used in explaining a relation with the defect. Most prominent combinations come from Size and Documentation metrics.

Studies in the domain point out very frequently the impact of the size of the code or code module and proneness to defect. One such study made an analysis that the size defect relationship is not a direct but rather a logarithmic relationship and that smaller modules (100 lines per class) tend to be more defective than larger (1000 lines

per class) [50] however in the context of cross-project defect prediction in general, it has been shown that large systems are bad predictors for small systems [51] hence other generalization efforts are done. That said, most maintainability efforts are placed on larger classes by developers [52]. Others have shown that there is a rather direct relationship between the quality of software and defects [53]. Destefanis et al [54] showed metric relevance throughout phases of an agile project to show that not only is the size of code impacting other metrics, but throughout multiple phases, its significance seems to spike up.

However, this information can be dependent on data as most project information used in this thesis indicates an overwhelmingly direct relationship between defect with size. For instance, The CKJM 1.8 dataset from the PROMISE dataset has an average of 219 lines of code per class for defective classes and 75 for non-defective, For Xerces 2.0, on average of 643 lines of code per class for defective modules and 175 for non-defective. This can be attributed to the fact that with changing software development practices, more lines of code are written in a shorter span of time and thus are subjected to testing where bugs are located. This is not to be used as some general rule unless we have a common framework to work in. The studies that show the impact of size have also considered release history as a factor. While we do not have historical data of one project, it was used implicitly to build a larger training data, rather than explicitly check how that information will impact results, especially if it could be used in a cross-project manner somehow.

The second aspect is the documentation metrics. Comments or API documentation are particularly useful in understanding the intended nature of a module and how to improve or fix it if functionality is not correct and leads to bugs. This, loosely speaking, encapsulates developer practices into the data as well and indicates that certain external factors do impact source code defects. No two developers will have the same style of coding. It will be an interesting problem to explore developer coding practices against a "correct" coding practice too. The dataset used however does not capture the details about the source code structure itself so the documentation-related information might be lacking.

A few features related to coupling, cohesion, and complexity were also used, with coupling one showing that relationships between classes or modules are also a factor that impacts defect finding.

However, more work could be done here. The selection techniques employed are few tailored for the type of data we used. Data distribution agnostic techniques and more exhaustive search methods used for every kind of model might produce different and more interesting results.

5.1.1 Response for RQ1.1

To define efficiency in terms of good accuracy, F1 Score, and time, we do not see a model that is good when compared to real-world requirements. Considering the data obtained in Section 4, a lot of information shows that the models are biased towards one label and are returning high false positives, something that is aimed to reduce. Model selection and feature selection methods surely impacted our results. However, the top efficient combination that has a potential for improvement, subject to tuning is

1. Voting classifier built with (RFC,XgB, SVC)
2. Voting classifier built with (RFC,XgB, kNN)
3. RFC with MutInfo features obtained via SMOTE
4. Multi-Layer Perceptron with Recursive feature elimination features

The aforementioned models have at least a precision greater than 0.5 and a prominent accuracy.

5.2 Response for RQ2

SMOTE by design generates synthetic samples that aim to oversample and reflect the minority class. The Tomek Links version is used which eliminates any collisions between the same feature and different labels. Between SMOTE and non-SMOTE versions, the feature sets do differ but as answered in RQ1, the common feature family still exists. The precision is notably increased and recall is more balanced. The validation test was performed as shown in Tables 4.8 and 4.9 show such. SMOTE did seem to improve performance as compared to other methods as it addressed the low precision obtained from non-SMOTE variants, however at the end, it is also subject to model selection. As a matter of fact, it is seen that SMOTE often decreases precision in favor of recall which was observed but the experiments in this thesis do not compare a single dataset and then subject it to SMOTE because of bias reduction purposes. The non-SMOTE variant is still largely impacted by high recall and thus appropriate tuning is needed. Because the voting classifier in the non-SMOTE version also has similarly high accuracy and F1 Score as its SMOTE counterpart, meaning that in summary SMOTE:

1. Improved F1 Score and detection of defective labels
2. Accuracy
3. Increased precision

However, this is just for this particular experiment design. The choice of dataset makes it difficult to compare with other similar studies but it is possible to use their experiment design on this dataset for comparison purposes.

Chapter 6

Conclusions and Future Work

In conclusion, the work done in this thesis and the results produced are not so profound. This was an exploratory finding of features from a wide array of metrics that relate to a software defect and a study on the impact of artificial data generation via SMOTE to see the impact.

We further learned and checked empirically how the size of the code makes an impact. It would be interesting to further delve into exactly what makes these seemingly basic metrics so important toward prediction quality. Aside from size, documentation is surprisingly an important aspect as seemingly non-code features are a really good aid/factor in our models. It seems that the basic instruction about always commenting on your code, offered in programming 101 classes, does hold some weight.

The study is lacking in the sense that comparison with other experimental designs could not be made fairly as the exact experimental setup would be needed. We compared some models with our own which used the same datasets or styles but that was not the major part of this work yet. A future plan for optimizing and tuning our models, creating a fair experimental lab for other similar studies, and comparing them, would be kept.

There are many shortcomings in this study however the dataset used is full of rich information on software measurements that an extensive and exhaustive look into understanding which feature sets are important and how better-tuned models can be made is interesting to ponder. As such, we have the following considerations for the future to improve upon and understand software defects better

1. Employ Exhaustive Searching methods in finding features. Instance-based learning may also be employed [31]

2. Metric results do not capture the internal code structure. Suppose we have 2 classes with similar cyclomatic complexity but one is buggy and the other is not, if we use SMOTE Tomek for instance, we lose key information.

3. Building on the last point, it is observed that numeric metric information is severely lacking in case of feature collision but the source code itself can provide the necessary difference to understand why a module is defective by providing structural and spatial details.

4. Move from numeric metrics to direct source code analysis. Studies have been done to show their usefulness over numeric metrics as structural and spatial information about source code is considered and the bug is tagged.

5. With the advent of better hardware and deep learning methods, studies have focused on source code directly to understand how its structure leads to defects. This arrangement, as per literature, has shown promise when dealing with cross-project defect prediction and will be interesting to explore further in this case too.

6. Explore how release-based or commit-based changes impact software defects. A study and collection of data by Martin et al. called Mega diff [55] is one comprehensive starting point.

7. There should be a standardized metric set to perform studies on. Most studies performed on this topic use specific data of an industry which becomes subject to privacy laws and Non-Disclosure agreements.

8. A combination of code structure and source code metrics can be used in cases where defect and non-defect data rows are similar, except for labels, as characteristics of those colliding rows are an interesting area to explore.

9. An interesting study, also mentioned in the literature review, about using change metrics to test defects. Source code metrics combined with change metrics might give valuable feedback when working on project defect prediction.

10. Understanding the impact of repetitive code. The code cloning metrics can also be used which are available in the dataset used in the thesis.

11. The development methodologies are not explicitly incorporated here. With methods like agile, DevOps, scrum, etc. it will be interesting to see how defects emerge in those particular environments.

Speaking in terms of experiment planning in the future

1. Create comparisons on a single dataset with models to check defect prediction.

2. Implement the experimental design of other studies to gauge performance and practically learn from results.

3. Use other SMOTE variants.

4. Explore the nature of modules with similar features but different labels.

5. Use other evaluation metrics that seek to explain and improve precision and recall.

6. Use different granularity of metrics to make predictions. For this Unified Bug Dataset version, we used Java projects, so object-oriented behavior is also assumed. Defect prediction in procedural code and relating to faults at a file level or statement is an interesting aspect to explore as done in other studies.

References

[1] Junting Gao, Liping Zhang, Fengrong Zhao, and Ye Zhai. Research on software defect classification. In *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, pages 748–754, 2019.

[2] Efi Papatheocharous, Stamatia Bibi, Ioannis Stamelos, and Andreas S. Andreou. An investigation of effort distribution among development phases: A four-stage progressive software cost estimation model. *Journal of Software: Evolution and Process*, 29(10):e1881, 2017. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1881.

[3] Fatih Gurcan, Gonca Gokce Menekse Dalveren, Nergiz Ercil Cagiltay, Dumitru Roman, and Ahmet Soylu. Evolution of software testing strategies and trends: Semantic content analysis of software research corpus of the last 40 years. *IEEE Access*, 10:106093–106109, 2022.

[4] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[5] Ahmed Abdu, Zhengjun Zhai, Redhwan Algabri, Hakim A. Abdo, Kotiba Hamad, and Mugahed A. Al-antari. Deep Learning-Based Software Defect Prediction via Semantic Key Features of Source Code—Systematic Survey. *Mathematics*, 10(17), 2022.

[6] Hongyu Zhang and Xiuzhen Zhang. Comments on "data mining static code attributes to learn defect predictors". *IEEE Transactions on Software Engineering*, 33(9):635–637, 2007.

[7] David Gray, David Bowes, Neil Davey, Yi Sun, and Bruce Christianson. Further thoughts on precision. In *15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011)*, pages 129–133, 2011.

[8] Emmanuel Gbenga Dada, David Opeoluwa Oyewola, Stephen Bassi Joseph, and AB Duada. Ensemble machine learning model for software defect prediction. *Adv. Mach. Learn. Artif. Intell*, 2:11–21, 2021.

[9] Cong Jin. Cross-project software defect prediction based on domain adaptation learning and optimization. *Expert Systems with Applications*, 171:114637, 2021.

[10] Le Hoang Son, Nakul Pritam, Manju Khari, Raghvendra Kumar, Pham Thi Minh Phuong, and Pham Huy Thong. Empirical study of software defect prediction: A systematic mapping. *Symmetry*, 11(2), 2019.

[11] David Bowes, Tracy Hall, and Jean Petrić. Software defect prediction: do different classifiers find the same defects? *Software Quality Journal*, 26(2):525–552, Jun 2018.

[12] D. Banthia and A. Gupta. A framework to assess the effectiveness of fault-prediction techniques for quality assurance. *CONSEG 2013 - Proceedings of the 7th CSI International Conference on Software Engineering*, pages 40–49, 01 2013.

[13] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks, 2019.

[14] Ran Li, Lijuan Zhou, Shudong Zhang, Hui Liu, Xiangyang Huang, and Zhong Sun. Software defect prediction based on ensemble learning. In *Proceedings of the 2019 2nd International Conference on Data Science and Information Technology*, DSIT 2019, page 1–6, New York, NY, USA, 2019. Association for Computing Machinery.

[15] Xiaozhi Du, Hehe Yue, and Honglei Dong. Software defect prediction method based on hybrid sampling. In *International Conference on Frontiers of Electronics, Information and Computation Technologies*, ICFEICT 2021, New York, NY, USA, 2022. Association for Computing Machinery.

[16] Anh Ho, Nguyen Nhat Hai, and Bui Thi-Mai-Anh. Combining deep learning and kernel pca for software defect prediction. In *Proceedings of the 11th International Symposium on Information and Communication Technology*, SoICT '22, page 360–367, New York, NY, USA, 2022. Association for Computing Machinery.

[17] Yihao Li, W. Eric Wong, Shou-Yu Lee, and Franz Wotawa. Using tri-relation networks for effective software fault-proneness prediction. volume 7, pages 63066–63080, 2019.

[18] Thanh Tung Khuat and My Hanh Le. Evaluation of sampling-based ensembles of classifiers on imbalanced data for software defect prediction problems. *SN Computer Science*, 1(2):108, Mar 2020.

[19] Jiehan Deng, Lu Lu, and Shaojian Qiu. Software defect prediction via lstm. *IET Software*, 14(4):443–450, 2020.

[20] Jian Li, Pinjia He, Jieming Zhu, and Michael R. Lyu. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–328, 2017.

[21] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, 108:115–138, 2019.

[22] Mattias Liljeson and Alexander Mohlin. Software defect prediction using machine learning on test and source code metrics, dissertation. 2014.

[23] Mark A. Hall. Correlation-based feature selection for machine learning. 1999.

[24] Rohit John Jacob, Rutuja J Kamat, N M Sahithya, Sharon Saji John, and Sahana P. Shankar. Voting based ensemble classification for software defect

prediction. In *2021 IEEE Mysore Sub Section International Conference (MysuruCon)*, pages 358–365, 2021.

[25] Sandro Morasca and Luigi Lavazza. On the assessment of software defect prediction models via roc curves. *Empirical Software Engineering*, 25(5):3977–4019, Sep 2020.

[26] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2019.

[27] Mahesha Bangalore Ramalinga Pandit and Nitin Varma. A deep introduction to ai based software defect prediction (sdp) and its current challenges. In *TENCON 2019 - 2019 IEEE Region 10 Conference (TENCON)*, pages 284–290, 2019.

[28] A Shanthini and R M Chandrasekaran. Analyzing the effect of bagged ensemble approach for software fault prediction in class level and package level metrics. In *International Conference on Information Communication and Embedded Systems (ICICES2014)*, pages 1–5, 2014.

[29] Md Nasir Uddin, Bixin Li, Md Naim Mondol, Md Mostafizur Rahman, Md Suman Mia, and Elizabeth Lisa Mondol. Sdp-ml: An automated approach of software defect prediction employing machine learning techniques. In *2021 International Conference on Electronics, Communications and Information Technology (ICECIT)*, pages 1–4, 2021.

[30] Hailemelekot Demtse Tessema and Surafel Lemma Abebe. Enhancing just-in-time defect prediction using change request-based metrics. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 511–515, 2021.

[31] Misha Kakkar and Sarika Jain. Feature selection in software defect prediction: A comparative study. pages 658–663, 01 2016.

[32] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. Deep learning based code smell detection. *IEEE Transactions on Software Engineering*, 47(9):1811–1837, 2021.

[33] Rudolf Ferenc, Zoltán Tóth, Gergely Ladányi, István Siket, and Tibor Gyimóthy. A public unified bug dataset for java and its assessment regarding metrics and bug prediction. *Software Quality Journal*, 28(4):1447–1506, Dec 2020.

[34] Görkem Giray, Kwabena Bennin, Omer Koksal, Önder Babur, and Bedir Tekinerdogan. On the use of deep learning in software defect prediction, 10 2022.

[35] Salsabila Ramadhina, Rizal Broer Bahawares, Irman Hermadi, Arif Imam Suroso, Ahmad Rodoni, and Yandra Arkeman. Software defect prediction using process metrics systematic literature review: Dataset and granularity level. In *2021 9th International Conference on Cyber and IT Service Management (CITSM)*, pages 1–7, 2021.

[36] Ivan Tomek. Two modifications of cnn. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-6(11):769–772, 1976.

[37] Yu Zhao, Yi Zhu, Qiao Yu, and Xiaoying Chen. Cross-project defect prediction considering multiple data distribution simultaneously. *Symmetry*, 14(2), 2022.

[38] Sourabh Pal and Alberto Sillitti. Cross-project defect prediction: A literature review. *IEEE Access*, 10:118697–118717, 2022.

[39] Archana Patnaik, Rasmita Panigrahi, and Neelamadhab Padhy. Prediction of accuracy on open source java projects using class level refactoring. In *2020 International Conference on Computer Science, Engineering and Applications (ICCSEA)*, pages 1–6, 2020.

[40] Abdullah Nasser, Waheed Ghanem, Antar Shaddad, Antar Abdul-Qawy, Mohammed Ali, Abdul-Malik Saad, Sanaa Ghaleb, and Nayef Alduais. A robust tuned k-nearest neighbours classifier for software defect prediction. 08 2022.

[41] Issam H. Laradji, Mohammad Alshayeb, and Lahouari Ghouti. Software defect prediction using ensemble learning on selected features. *Information and Software Technology*, 58:388–402, 2015.

[42] Monalisa Ghosh and Goutam Sanyal. An ensemble approach to stabilize the features for multi-domain sentiment analysis using supervised machine learning. *Journal of Big Data*, 5(1):44, Nov 2018.

[43] Fei Wu, Xiao-Yuan Jing, Xiwei Dong, Jicheng Cao, Mingwei Xu, Hongyu Zhang, Shi Ying, and Baowen Xu. Cross-project and within-project semi-supervised software defect prediction problems study using a unified solution. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 195–197, 2017.

[44] Yiwen Zhong, Kun Song, ShengKai Lv, and Peng He. An empirical study of software metrics diversity for cross-project defect prediction. *Mathematical Problems in Engineering*, 2021:1–11, 11 2021.

[45] Jiehan Deng, Lu Lu, Shaojian Qiu, and Yangpeng Ou. A suitable ast node granularity and multi-kernel transfer convolutional neural network for cross-project defect prediction. *IEEE Access*, PP:1–1, 04 2020.

[46] Wanzhi Wen, Ruinian Zhang, Chuyue Wang, Chenqiang Shen, Meng Yu, Suchuan Zhang, and Xinxin Gao. A cross-project defect prediction model based on deep learning with self-attention. *IEEE Access*, PP:1–1, 01 2022.

[47] Wei Zheng, Lijuan Tan, and Chengbin Liu. Software defect prediction method based on transformer model. In *2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*, pages 670–674, 2021.

[48] Milind Shah, Kinjal Gandhi, Kinjal A Patel, Harsh Kantawala, Rohini Patel, and Ankita Kothari. Theoretical evaluation of ensemble machine learning techniques. In *2023 5th International Conference on Smart Systems and Inventive Technology (ICSSIT)*, pages 829–837, 2023.

[49] Faseeha Matloob, Taher M. Ghazal, Nasser Taleb, Shabib Aftab, Munir Ahmad, Muhammad Adnan Khan, Sagheer Abbas, and Tariq Rahim Soomro. Software

defect prediction using ensemble learning: A systematic literature review. *IEEE Access*, 9:98754–98771, 2021.

[50] A. Güneş Koru, Dongsong Zhang, Khaled El Emam, and Hongfang Liu. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering*, 35(2):293–304, 2009.

[51] Anushree Agrawal and Ruchika Malhotra. Cross project defect prediction for open source software. *International Journal of Information Technology*, 14:1–15, 04 2019.

[52] Péter Hegedűs, István Kádár, Rudolf Ferenc, and Tibor Gyimóthy. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Information and Software Technology*, 95:313–327, 2018.

[53] Zhizhong Jiang, Peter Naudé, and Binghua Jiang. The effects of software size on development effort and software quality. volume 34, 01 2007.

[54] Giuseppe Destefanis, Giulio Concas, Michele Marchesi, and Roberto Tonelli. An empirical study of software metrics for assessing the phases of an agile project. *International Journal of Software Engineering and Knowledge Engineering*, 22, 06 2012.

[55] Martin Monperrus, Matias Martinez, He Ye, Fer Madeiral, Thomas Durieux, and Zhongxing Yu. Megadiff: A dataset of 600k java source code changes categorized by diff size, 08 2021.

Appendix A

Supplemental Information

A.1 Full form and definition of metrics

LCOM5 - Lack of Cohesion in Methods 5 - Measures cohesiveness of class members.

cam - Cohesion among methods of class - relatedness of methods by virtue of parameter list

cbm - coupling between methods - Number of new or redefined methods to which all inherited methods are coupled

NL - Nesting Level - depth of the maximum embeddedness of its conditional - iteration and exception handling block scopes

NLE - Nesting Level If Else - depth of the maximum embeddedness of its conditional - iteration and exception handling block scopes - where in the if-else-if construct only the first if the instruction is considered.

wmc - weighted method per class - the number of independent control flow paths in it

max_cc - maximum McCabe cyclomatic complexity - maximum value for McCabe cyclomatic complexity

avg_cc - average McCabe cyclomatic complexity - average value for McCabe cyclomatic complexity

mfa - measure of functional abstraction - Number of methods inherited by a class per number of methods accessible by ite methods

dam - data access metric - Ratio of number of private and protected attributes to ttal attributes

CBOI - Coupling between object classes - number of other classes - which directly use the class.

NII - Number of incoming invocations - number of other methods and attribute initializations that directly call the local methods of the class.

NOI - Number of outgoing invocations - number of directly called methods of other classes - including method invocations from attribute initialization. If a method is invoked several times - it is counted only once.

cbo - coupling between objects - number of directly used other classes (e.g.by inheritance - function call - type reference - attribute reference).

rfc - response for class - number of local (i.e.not inherited) methods in the class (NLM) plus the number of directly invoked other methods by its methods or attribute initialization

ca - Afferent coupling - Number of other classes which depend on the class

ce - Efferent Coupling - Number of other classes on which the class depends

moa - Measure of aggregation - extent of relationshp realized by attributes of class

AD - API Documentation - ratio of the number of documented public methods in the class +1 if the class itself is documented to the number of all public methods in the class + 1 (the class itself); however - the nested - anonymous - and local classes are not included.

CD - Comment Density - ratio of the comment lines of the class (CLOC) to the sum of its comment (CLOC) and logical lines of code (LLOC)

CLOC - Comment Lines of Code - number of comment and documentation code lines of the class - including its local methods and attributes; however - its nested - anonymous - and local classes are not included.

DLOC - Documentation Line of Code - number of documentation code lines of the class - including its local methods and attributes; however - its nested - anonymous - and local classes are not included.

PDA - Public Documented API - number of documented public methods in the class (+1 if the class itself is documented); however - the methods of its nested - anonymous - and local classes are not counted.

PUA - Public Undocumented API - number of undocumented public methods in the class (+1 if the class itself is undocumented); however - the methods of its nested - anonymous - and local classes are not counted.

TCD - Total Comment Density - ratio of the total comment lines of the class (TCLOC) to the sum of its total comment (TCLOC) and total logical lines of code (TLLOC)

TCLOC - Total Comment Line of Code - number of comment and documentation code lines of the class - including its local methods and attributes - as well as its nested - anonymous - and local classes.

DIT - Depth of Inheritance Tree - length of the path that leads from the class to its farthest ancestor in the inheritance tree.

NOA - Number of Ancestors - number of classes - interfaces - enums and annotations from which the class is directly or indirectly inherited.

NOD - Number of Descendants - number of classes - interfaces - enums - annotations - which are directly or indirectly derived from the class.

NOP - Number of Parents - number of classes - interfaces - enums and annotations from which the class is directly inherited.

ic - inheritance coupling - Number of parent classes to which a given class is coupled

noc - Number of Children - number of classes - interfaces - enums and annotations which are directly derived from the class.

LLOC - Logical Lines of Code - number of non-empty and non-comment code lines of the class - including the non-empty and non-comment lines of its local methods; however - its nested - anonymous - and local classes are not included.

NA - Number of attributes - number of attributes in the class - including the inherited ones; however - the attributes of its nested - anonymous - and local classes are not included.

NG - Number of getters - number of getter methods in the class - including the inherited ones; however - the getter methods of its nested - anonymous - and local classes are not included. Methods that override abstract methods are not counted.

NLA - Number of local attributes - number of local (i.e.not inherited) attributes in the class; however - the attributes of nested - anonymous - and local classes are not included.

NLG - Number of local getters - number of local (i.e.not inherited) getter methods in the class; however - the getter methods of its nested - anonymous - and local classes are not included. Methods that override abstract methods are not counted.

NLM - Number of local methods - number of local (i.e.not inherited) methods in the class; however - the methods of nested - anonymous - and local classes are not included.

NLPA - Number of local public attributes - number of local (i.e.not inherited) public attributes in the class; however - the attributes of nested - anonymous - and local classes are not included.

NLPM - Number of local public methods - number of local (i.e.not inherited) public methods in the class; however - the methods of nested - anonymous - and local classes are not included.

NLS - Number of local setters - number of local (i.e.not inherited) setter methods in the class; however - the setter methods of its nested - anonymous - and local classes are not included. Methods that override abstract methods are not counted.

NM - Number of methods - number of methods in the class - including the inherited ones; however - the methods of its nested - anonymous and local classes are not included. Methods that override abstract methods are not counted.

NOS - Number of statements - number of statements in the method; however - the statements of its anonymous and local classes are not included.

NPA - Number of public attributes - number of public attributes in the class - including the inherited ones; however - the public attributes of its nested - anonymous - and local classes are not included.

NS - Number of setters - number of setter methods in the class - including the inherited ones; however - the setter methods of its nested - anonymous - and local classes are not included. Methods that override abstract methods are not counted.

TLLOC - Total logical lines of code - number of non-empty and non-comment code lines of the class - including the non-empty and non-comment code lines of its local methods - anonymous - local - and nested classes.

TLOC - total lines of code - number of code lines of the class - including empty and comment lines - as well as its local methods - anonymous - local - and nested classes.

TNA - total number of attributes - number of attributes in the class - including the inherited ones - as well as the inherited and local attributes of its nested - anonymous and local classes.

TNG - total number of getters - number of getter methods in the class - including the inherited ones - as well as the inherited and local getter methods of its nested - anonymous and local classes.

TNLA - Total Number of Local Attributes - number of local (i.e.not inherited) attributes in the class - including the attributes of its nested - anonymous - and local classes.

TNLG - Total Number of Local Getters - number of local (i.e.not inherited) getter methods in the class - including the local getter methods of its nested - anonymous - and local classes.

TNLM - Total Number of Local Methods - number of local (i.e.not inherited) methods in the class - including the local methods of its nested - anonymous - and local classes.

TNLPA - Total Number of Local Public Attributes - number of local (i.e.not inherited) public attributes in the class - including the local public attributes of its nested - anonymous - and local classes.

TNLPM - Total Number of Local Public Methods - number of local (i.e.not inherited) public methods in the class - including the local methods of its nested - anonymous - and local classes

TNLS - Total Number of Local Setters - number of local (i.e.not inherited) setter methods in the class - including the local setter methods of its nested - anonymous - and local classes.

TNM - Total Number of Methods - number of methods in the class - including the inherited ones - as well as the inherited and local methods of its nested - anonymous - and local classes. Methods that override abstract methods are not counted

TNOS - Total Number of Statements - number of statements in the class - including the statements of its nested - anonymous - and local classes

TNPA - Total Number of Public Attributes - number of public attributes in the class - including the inherited ones - as well as the inherited and local public attributes of its nested - anonymous - and local classes.

TNPM - Total Number of Public Methods - number of public methods in the class - including the inherited ones - as well as the inherited and local public methods of its nested - anonymous - and local classes.

TNS - Total Number of Setters - number of setter methods in the class - including the inherited ones - as well as the inherited and local setter methods of its nested - anonymous and local classes.

npm - number of public methods - Number of public methods of a class

loc - line of code - number of code lines of the class - including empty and comment lines - as well as its local methods; however - its nested - anonymous - and local classes are not included.

amc - average method complexity - average method size for each class

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden