Western Graduate&PostdoctoralStudies

Western University
Scholarship@Western

Electronic Thesis and Dissertation Repository

12-19-2018 2:30 PM

Predicting Software Fault Proneness Using Machine Learning

Sanjay Ghanathey
The University of Western Ontario

Supervisor
Konstantinos Kontogiannis
The University of Western Ontario

Graduate Program in Computer Science
A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science
© Sanjay Ghanathey 2018

Follow this and additional works at: https://ir.lib.uwo.ca/etd

Part of the Software Engineering Commons

Recommended Citation
Ghanathey, Sanjay, "Predicting Software Fault Proneness Using Machine Learning" (2018). *Electronic Thesis and Dissertation Repository*. 5936.
https://ir.lib.uwo.ca/etd/5936

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlswadmin@uwo.ca.

Abstract

Context: Continuous Integration (CI) is a DevOps technique which is widely used in practice. Studies show that its adoption rates will increase even further [38]. At the same time, it is argued that maintaining product quality requires extensive and time consuming, testing and code reviews. In this context, if not done properly, shorter sprint cycles and agile practices entail higher risk for the quality of the product. It has been reported in literature [68], that lack of proper test strategies, poor test quality and team dependencies are some of the major challenges encountered in continuous integration and deployment.

Objective: The objective of this thesis, is to bridge the process discontinuity that exists between development teams and testing teams, due to continuous deployments and shorter sprint cycles, by providing a list of potentially buggy or high risk files, which can be used by testers to prioritize code inspection and testing, reducing thus the time between development and release.

Approach: Out approach is based on a five step process. The first step is to select a set of systems, a set of code metrics, a set of repository metrics, and a set of machine learning techniques to consider for training and evaluation purposes. The second step is to devise appropriate client programs to extract and denote information obtained from GitHub repositories and source code analyzers. The third step is to use this information to train the models using the selected machine learning techniques. This step allowed to identify the best performing machine learning techniques out of the initially selected in the first step. The fourth step is to apply the models with a voting classifier (with equal weights) and provide answers to five research questions pertaining to the prediction capability and generality of the obtained fault proneness prediction framework. The fifth step is to select the best performing predictors and apply it to two systems written in a completely different language (C++) in order to evaluate the performance of the predictors in a new environment.

Obtained Results: The obtained results indicate that a) The best models were the ones applied on the same system as the one trained on; b) The models trained using repository metrics outperformed the ones trained using code metrics; c) The models trained using code metrics were proven not adequate for predicting fault prone modules; d) The use of machine learning as a tool for building fault-proneness prediction models is promising, but still there is work to be done as the models show weak to moderate prediction capability.

Conclusion: This thesis provides insights into how machine learning can be used to predict whether a source code file contains one or more faults that may contribute to a major system failure. The proposed approach is utilizing information extracted both from the system's source code, such as code metrics, and from a series of DevOps tools, such as bug repositories, version control systems and, testing automation frameworks. The study involved five Java and five Python systems and indicated that machine learning techniques have potential towards building models for alerting developers about failure prone code.

Keywords: Software analysis, code metrics, repository metrics, software metrics, Software defect analysis

Acknowledgements

First, I would like to express my profound gratitude to my advisor Dr. Kostas Kontogiannis, for the continuous support of my master's program and research, for his patience, motivation, and immense knowledge. It was his supervision and guidance that helped me to decide my research path and walk through it. I have been extremely lucky to have a supervisor who cared so much, planned regular meetings, detailed explanation and discussions ensured that I was on the right track. I will forever be thankful to Dr. Kostas Kontogiannis, without whom I would have not been where I am today.

I am grateful to IBM (International Business Machines Corporation) for giving me the opportunity to do part of my research in their premises through my graduate program. My sincere thanks to Chris Brealey and Alberto Giammaria at IBM who not only guided me throughout the internship but also in my Master's research. I would also like to thank my fellow lab mates, Marios-Stavros Grigoriou, Hao Jiang, Darlan Arruda and Konstantinos Tsiounis for the amazing discussions during our group meetings, project work, presentations, and for all the fun we have had in the lab. It was our lab environment that made the whole research smooth and thought provoking. Last but not the least, I would like to thank my family for supporting me spiritually throughout my master's and life in general.

Contents

Abstract i

Acknowledgements ii

List of Figures vi

List of Tables viii

1 Introduction 1
- 1.1 Introduction 1
  - 1.1.1 Problem Area and Motivation 2
  - 1.1.2 Contributions, Scope and limitations 2
  - 1.1.3 Research questions 3
  - 1.1.4 Thesis outline 4

2 Background and related work 5
- 2.1 Background 5
  - 2.1.1 Code Metrics 5
  - 2.1.2 Machine Learning 6
  - 2.1.3 Technical Debt 9
    - Architecture Debt 9
    - Build Debt 9
    - Code Debt 10
    - Defect Debt 10
    - Design Debt 10
    - Documentation Debt 10
    - Test Debt 11
    - Requirement Debt 11
    - Infrastructure Debt 11
    - People Debt 11
  - 2.1.4 GitHub 12
- 2.2 Related Work 13
  - 2.2.1 Software Bug Prediction using Machine Learning 13
  - 2.2.2 Software Metrics 16
    - Object Oriented Metrics 16
    - Repository Metrics 17

Test Metrics . . . 18
Dependency Metrics . . . 18
Hybrid metrics . . . 18
Technical Debt Metrics . . . 19
2.2.3 Results Comparison . . . 20

3 Fault-proneness Prediction 21
3.1 System Selection . . . 21
3.1.1 Criteria . . . 21
3.1.2 Method . . . 22
3.1.3 Selected Systems . . . 22
3.2 Metric Selection . . . 24
3.2.1 Repository metrics . . . 24
3.2.2 Code metrics . . . 25
3.3 Data Collection Process . . . 26
3.3.1 Identify files with errors . . . 29
3.3.2 Collection of repository information and metrics . . . 30
3.3.3 Collection of static code metrics . . . 30
3.3.4 Experimentation Framework . . . 30
3.4 Building a model . . . 30
3.4.1 Model . . . 30
3.4.2 Re-sampling and highly skewed features . . . 31
3.4.3 Model evaluation . . . 31

4 Analysis and Obtained Results 33
4.1 Research Questions RQ1a and RQ1b . . . 33
4.2 Research Question RQ2 . . . 34
4.3 Research Question RQ3 . . . 36
4.4 Research Question 4 . . . 37
4.5 Research Question 5 . . . 41

5 Additional Model Validation in C++ Systems 44
5.1 Model Validation Overview . . . 44
5.2 Results . . . 45
5.2.1 Using model from RQ3 . . . 45
5.2.2 Using models from RQ2: . . . 45

6 Discussion, Future Work and Conclusion 48
6.1 Discussion . . . 48
6.1.1 Thesis Findings . . . 50
6.1.2 Threats to validity . . . 50
6.2 Future Work . . . 51
6.2.1 Additional metrics and projects . . . 51
6.2.2 Advanced machine learning models . . . 51
6.2.3 Technical Debt . . . 52

6.3 Conclusion 55

Bibliography **56**

A Data Extraction Algorithms **63**
A.1 Data Extraction Algorithms: 64
A.1.1 Algorithm to extract Bug or Issue Data from GitHub: 64
A.1.2 Algorithm to combine Bug or Issue Data with metrics: 66
A.1.3 Algorithm - Machine Learning: 69

B Machine Learning - Detailed Report **77**
B.1 Cross Validation Scores: 78
B.2 Detailed Report: 78
B.2.1 RQ1: 78
Repository metrics: 78
Code metrics: 84
B.2.2 RQ2: 89
Repository metrics: 89
Code metrics: 90
B.2.3 RQ3: 91
Repository metrics: 91
Code metrics: 92
B.2.4 RQ4: 93
Repository metrics: 93
Code metrics: 94
B.2.5 Validation: 95
RQ2: 95

Curriculum Vitae **97**

List of Figures

2.1 Overview of Github's [1] API Data Model . . . 12

3.1 Overview of project selection, research questions, experiment and validation . . . 23
3.2 Fractal visualization example [77] . . . 24
3.3 Sum of Coupling visualization example [77] . . . 25
3.4 Data collection from [1] using metrics tools [4, 8]. . . . 27
3.5 Sample data with repository metrics. . . . 29
3.6 Sample data with code metrics. . . . 29
3.7 Imbalanced classes for springboot project . . . 31

4.1 Advanced Code Metrics . . . 41

5.1 Kopete and K3b detailed reports . . . 46

6.1 Code Debt example [73] . . . 53
6.2 Code Debt example [73] . . . 54

B.1 Overview of project selection, research questions, experiment and validation . . . 78
B.2 Springboot project . . . 79
B.3 Deeplearning4j project . . . 79
B.4 Elasticsearch project . . . 80
B.5 Eclipse-che project . . . 80
B.6 RxJava project . . . 81
B.7 Youtube-dl project . . . 81
B.8 Ipython project . . . 82
B.9 Scikit project . . . 82
B.10 Scrapy project . . . 83
B.11 Keras project . . . 83
B.12 Springboot project . . . 84
B.13 Deeplearning4j project . . . 84
B.14 Elasticsearch project . . . 85
B.15 Eclipse-che project . . . 85
B.16 RxJava project . . . 86
B.17 Youtube-dl project . . . 86
B.18 Ipython project . . . 87
B.19 Scikit project . . . 87
B.20 Scrapy project . . . 88
B.21 Keras project . . . 88

B.22 Repository metrics on java projects only 89
B.23 Repository metrics on python projects only 89
B.24 Code metrics on java projects only 90
B.25 Code metrics on python projects only 90
B.26 Repository metrics with java as test project 91
B.27 Repository metrics with python as test project 91
B.28 Code metrics with java as test project 92
B.29 Code metrics with python as test project 92
B.30 Repository metrics with java as test project 93
B.31 Repository metrics with python as test project 93
B.32 Code metrics with java as test project 94
B.33 Code metrics with python as test project 94
B.34 Kopete detailed report using Python projects 95
B.35 K3b detailed report using Python projects 95
B.36 Kopete detailed report using Java projects 96
B.37 K3b detailed report using Java projects 96

List of Tables

2.1 Confusion Matrix 9

3.1 Systems used in the experiment. 23

4.1 Repository metrics Before re-sampling and After re-sampling. 34
4.2 Code Metrics Before re-sampling and After re-sampling. 35
4.3 Example for Aggregated method. 35
4.4 Superimposed (Repository and code metrics) - Before and After re-sampling. . 36
4.5 Predicting for Java project (springboot) and Python project (scikit) using repository metrics. 37
4.6 Predicting for Java project (springboot) and Python project (scikit) using code metrics. 38
4.7 Predicting for Java project (springboot) and Python project (ipython) using repository metrics. 38
4.8 Predicting for Java project (springboot) and Python project (ipython) using code metrics. 39
4.9 Predicting bugs by partitioning each project of same language into train and test data. 40
4.10 Advanced Code Metrics 42
4.11 Predicting for Java project (springboot) and Python project (scikit) using advanced code metrics. 42
4.12 Predicting for Java project (springboot) and Python project (ipython) using advanced code metrics. 43
4.13 Predicting bugs by partitioning each project of same language into train and test data. 43

5.1 Systems used in validating the experiment. 44
5.2 Comparing results of data validation on C++ projects with RQ3 repository metrics results. 45
5.3 Comparing results of data validation on C++ projects with RQ2 repository metrics results. 46
5.4 Comparing results of data validation on C++ projects with RQ2 repository metrics results. 47

6.1 Summary of the discussion 49

Chapter 1

Introduction

1.1 Introduction

Before any software application is delivered to the clients, it has to be tested so that it is verified that it meets its functional and non-functional requirements. Software testing is often paired with software reliability prediction models, so that the overall testing time required for the system to reach a specific failure intensity level can be estimated. However, for many non-mission critical applications which nevertheless have strict time to market constraints or the strict release deadlines, we often require the use of a triage method in order to prioritize testing depending on the estimated risk of failure of each software module on the application being tested. For many systems (e.g. mobile applications) short release cycles make the testing of every possible execution path not feasible, while meeting at the same time strict release deadlines. In this respect, we need to devise a technique by which we can alert both developers and testers regarding the risk of failure of a software component. This will help to appropriately schedule and prioritize the test cases to be applied. In this thesis, we conduct and report results from a series of experiments in order *a)* to investigate whether machine learning can be a feasible approach towards developing models for evaluating the fault proneness of a software module and *b)* identify a number of source code and non-source code related features which may be of use towards training the aforementioned models.

The main focus of this thesis is to devise fault prediction models, so that software developers and software testers can obtain an initial view on how to manage and prioritize testing in large software systems. More specifically, given a software system $\mathcal{S}$, a commit action C_i involving a collection of revised files $R_i = \{F_{1,i}, F_{2,i}, \ldots F_{k,i}\}$, and a trained model $\mathcal{M}$ on a history of revisions and system metrics $\mathcal{H}$, we would like to be able to predict which of the files $F_{j,i}$ are fault prone, so that a) developers can be alerted for possible impeding failures of the file(s) they have revised; b) testers can prioritize and re-schedule their test plans so that fault prone modules can be tested first, and; c) managers can be alerted for possible failure risks. The system can serve as a first step on a larger pipeline that supports continuous software engineering (integration, release, and deployment) by providing insights for automating DevOps processes.

More specifically, in this thesis we present a framework for training and applying machine learning models for the purpose of evaluating the likelihood of a file being fault prone. In this study we use source code metrics as well as non-source code information obtained from vari-

ous repositories (e.g. number and frequency of commits, past failures, refactoring operations), in order to train the machine learning framework. In this thesis, and according to IEEE terminology [63] we use the term *fault* to indicate an incorrect step, process, or data definition in a computer program (i.e a bug). A *fault* may cause an *error* which is a difference between a computed result and a correct result and it is internal to the state of the program. Consequently, an *error* may cause a *failure* which is a deviation of the expected behavior of the program from the *observed* one.

In this thesis, we have collected data from five Java and five Python open source systems that are available on GitHub[1], as illustrated in Table 3.1. We have selected parts of these systems to train a number of fault proneness prediction models, and we have consequently applied these models on the rest of the aforementioned systems. As an additional validation and in order to prove the generality of the prediction models, we have applied the best performing prediction models on two large open source C++ systems obtained from the KDE repositories.

1.1.1 Problem Area and Motivation

Problem Area: Software Analytics for the identification and assessment of fault proneness of a software system at the file level.

Motivation: Most of the software applications are almost certain to produce failures due to faults (bugs) introduced during the development or maintenance phases. A good number of these errors are discovered by the testing team and few of them are discovered over time in the operational phase.

However, it has been reported that the cost of fixing a bug in later stages of the software development cycle can be very expensive when compared to the development phase [70, 15]. Therefore, one must take this into account and try to find and fix bugs as early as possible.

The simplest way to find bugs is by testing. In [53] software testing is defined as ‘the process of executing a program with the intent of finding errors‘. There are different types of testing as described in [13] such as unit testing, function testing, system testing and integration testing.

It has been observed that most of the time the testing activities consume anything between 45% to 75% of the total development time [25]. By providing predictions on whether a file contains a fault or not, we may be able to reduce the amount of time consumed during the testing phase by focusing first on the probable buggy files. With respect to bug prediction, there have been many studies since 1970’s ranging from simple equations for measuring code complexities [30] to machine learning algorithms [20, 35].

1.1.2 Contributions, Scope and limitations

Even though significant work has already been conducted in the area of using Machine Learning for software fault prediction, to-date there are no authoritative models that can be used to predict fault proneness in large software systems. More specifically, software engineers and computer scientists are still experimenting with different Machine Learning frameworks as well as source code and repository features in a quest to identify the best frameworks to use along

with a standardized set of features and a standardized set of pre-processed data so that these can be used to train such Machine Learning frameworks in order to yield predictive models. This thesis falls in the area of Experimental Software Engineering, and aims to shed light in the problem of identifying such a collection of Machine Learning frameworks, repository features, and software metrics that can be used for the generation of fault prediction models. The objective is to provide experimental data points for software engineers and computer scientists to use towards developing more and more accurate predictive models. The thesis focuses on the analysis of ten large open source systems by considering a volume of more than 75,000 issue-bearing commit records. The thesis contributions can be summarized as follows:

- Investigate and report on the effectiveness of selected features and metrics to be used for language agnostic software fault proneness prediction at the file level;
- Investigate different Machine Learning frameworks and their effectiveness on yielding fault prediction models, and;
- Investigate five key research questions RQ1-RQ5 (please see section below) that can shed light on the effectiveness of the predictive models under different operational scenarios.

The thesis is comprised of:

- A tool for extracting repository metrics [8] from GitHub.
- A tool for extracting source code metrics for Java and Python projects [4].
- Custom made Java application programs (Appendix A.1.1,A.1.2) for combining the metrics and bug reports into a desired format.
- Use of existing open source infrastructure provided by Kaggle [2] in conjunction with the use of custom made programs to train and evaluate prediction models.
- The analysis, discussion, and explanation of the obtained results.

This thesis, however, does not address whether the models have similar performance to different types of applications (e.g real-time systems, enterprise systems, systems that depend on scripting languages). Furthermore, the thesis does not take into account lower level source code dependencies or other low level source code related information (e.g. information extracted from the AST).

1.1.3 Research questions

In this work we aim to find and report on empirical evidence to answer the following research questions, which constitute a typical set of research question found in the related literature.

RQ1a: Is it possible to train models that can be used to predict fault-proneness of individual projects, by using repository and source code metrics?

RQ1b: With respect to results from **RQ1a** does the choice of the programming language affect the results?

RQ2: Given a set of projects $\{P_1, P_2, P_3...., P_n\}$, written in a language L, is it possible to train the model on all projects $\{P_1, P_2, P_3...., P_{n-1}\}$ in order to predict fault proneness for project P_n?

RQ3: Given a set of projects $\{P_1, P_2, P_3...., P_n\}$, written in languages $\mathcal{L} = \{L_1, L_2, L_3...., L_k\}$ is it possible to train the model on all projects $\{P_1, P_2, P_3...., P_{n-1}\}$ in order to predict bugs for project P_n written in a language $L_i \in \mathcal{L}$?

RQ4: Given a set of projects $\mathcal{P}$=$\{P_1, P_2, P_3...., P_n\}$, written in the same language L, is it possible to predict fault proneness of a project $P_i \in \mathcal{P}$ by partitioning the projects $\{P_1, P_2, P_3...., P_n\}$ into two sets, one set (80%) for training the model and the other set (20%) for predicting fault proneness?

RQ5: Using advanced code metrics, answer RQ1, RQ2, RQ3 and RQ4. Do advanced code metrics improve the results?

1.1.4 Thesis outline

The remainder of this thesis is organized as follows. Chapter 2 provides background information that is useful in placing this work in context. Chapter 2 also provides information on related work describing different predictor models proposed in the literature, as well as evaluation metrics and strategies. In Chapter 3 we discuss in detail the repository metrics, code metrics, machine learning framework, as well as the modeling and the infrastructure used for our evaluation study. In Chapter 4 we report on the results obtained by using the trained models, while in Chapter 5, we validate our model (generated using GitHub projects - in the Java and Python programming languages) on KDE projects (C++ programming language). Finally, in Chapter 6 we provide an interpretation of the results, provide pointers for future research, and we conclude the thesis.

Chapter 2

Background and related work

2.1 Background

2.1.1 Code Metrics

Static code metrics are metrics which can be extracted directly from the source code, such as the total number of lines of code (LOC), Halstead metrics, Henry-Kafura metrics, and cyclomatic complexity metrics. One such static code metric is the Source Lines of Code (SLOC) which represents a related family of metrics focusing on counting lines in a source code file. Among others, the SLOC family includes, the total number of blank lines (BLOC), the total number of lines in a file (LOC), the total number of commented lines of code (CLOC), and the total number of the logical lines of source code (SLOC-L) which is the total number of executable lines of code [57, 5, 6]. The Cyclomatic Complexity (CCN), also known as the McCabe metric or the McCabe Cyclomatic Complexity, is a measure of the complexity of the decision structure, in the control flow graph of the code of a module. The metric introduced by Thomas McCabe [49] and is equal to the number of linearly independent paths in the control flow graph [49].

SR Chidamber et al. in [23] and J Bansiya et al. in [12] discuss object oriented metrics such as the:

1. *Weighted Method per class (WMC)*, which is the number of methods in the class [23].

2. *Depth of inheritance Tree(DIT)* which measures the inheritance levels from the top of the hierarchy of objects for each class. [23].

3. *Number of Children (NOC)* which measures the number of the class's immediate descendants [23].

4. *Coupling between Objects (CBO)* which is a metric that represents the number of classes coupled to a class. The coupling can be achieved through method calls, field accesses, inheritance, method arguments, return types and exceptions. [23].

5. *Response for a class (RFC)* which is the number of Distinct Methods and Constructors invoked by a class [23].

6. *Lack of Cohesion of methods (LCOM):* If 'm' is the total number of *methods* in a class and 'a' is the total number of *attributes* in the class then LCOM is computed as follows: 1 – (sum(am)/a*m) where 'am' is the number of methods accessing a particular attribute and 'sum(am)' is the total sum of 'am' over all the instances in the class [23, 3].

7. *Number of Public Methods (NPM)* which is a metric that counts all the methods in a class that are declared as public [12].

8. *Data Access Metric (DAM)* which is a metric that is the ratio of the number of private attributes to the total number of attributes declared in the class [12].

9. *Measure of Aggregation (MOA)* which is a metric that measures the extent of the part-whole relationship (aggregation, composition), realized by using attributes. The metric is a count of the number of class fields whose types are user defined classes [12].

10. *Measure of Functional Abstraction (MFA)* which is the ratio of the number of methods inherited by a class to the total number of methods accessible by the member methods of the class [12].

11. *Cohesion Among Methods of Class (CAM)* which computes the relatedness among methods of a class based upon the parameter list of the methods. The metric is computed using the summation of a number of different types of method parameters in every method, divided by a multiplication of number of different method parameter types in whole class and number of methods [12].

2.1.2 Machine Learning

Machine Learning is a branch of Artificial Intelligence concerning computer programs learning from data. Machine Learning aims at imitating the human learning process with computers, and is all about observing a phenomenon and learning from the observations [42]. Machine Learning can be broadly divided into two categories: supervised and unsupervised learning. Supervised learning concerns learning from examples with known outcome for each of the training samples [82], while unsupervised learning tries to learn from data without known outcome. Supervised learning is sometimes called classification, as it classifies instances into two or more classes. It is the classification problem that this thesis focuses on, since based on the features of the file, the machine learning algorithm predicts whether a file is buggy or not. There is a variety of machine learning algorithms for classification (supervised learning). These families includes adaboost, decision trees [44], support vector machines (SVM) [79, 33], neural networks [91, 75] and deep learning, random forest[16], k-nearest neighbors (KNN), extra trees, logistic regression and gradient boosting. Ayodele in [11] classifies supervised learning which deals with classification as Linear Classifiers, Quadratic Classifiers, K-Means Clustering, Boosting, Decision Tree, Neural networks and Bayesian Networks. Ayodele [11] further classifies Linear Classifiers into four types - Logical Regression, Naïve Bayes Classifier, Perceptron and Support Vector Machine(SVM).

Support vector machine for classification (SVC): Support vector machines (SVM) have been proposed by Vapnik [79] along with other researchers, and they have been widely studied and applied in many fields. The basic idea of SVM is to to identify a similarity distance between two entities (classes) by considering a distance metric between them. The distance between such entities (classes) is traditionally defined as a function of their feature vectors [39]. Support Vector Machine could also be used to accommodate for unbalanced classes[61]

Decision tree: Decision tree classifiers use comparisons to divide different instances of a set into appropriate classes. *Classification* is a type of supervised learning where initially, a set of known instances, called the training set, is introduced to a system. The system classifies each instance of the set, associates each class with the attributes of each instance and learns to what class each instance belongs. Based on what the trained system has learned in the learning phase, it is able to classify instances of a previously unseen set [74].

One application of using Decision trees is for pattern recognition algorithms [74]. Another application of this type of Machine Learning is image analysis such as in cancer cell and brain tumor detection [21].

AdaBoost: As described in [10], the AdaBoost algorithm is one of the most well-known algorithms for building an *ensemble* classifier. Each instance in the training dataset is weighted. The initial weight is set to: weight(x_i) = 1/n, where x_i is the ith training instance and n is the number of training instances. The most common algorithm used with AdaBoost are *decision trees* with one level. As these trees are short and contain only one decision for classification, they are called as *decision stumps*. A weak classifier (*decision stump*) is prepared on the training data using the weighted samples. The AdaBoost algorithm generates a strong classifier by adjusting weights through a repetition process[24].

Random forest: Breiman [16] suggested a new and promising classifier in (2001) called random forest, which presents many advantages [32] such as its running efficiently on large databases, being able to handle thousands of input variables, and providing estimates indicating which variable is important in a classification session.

K Nearest Neighbors (KNN): As discussed in [41], among the various methods for supervised statistical pattern recognition, the Nearest Neighbor rule achieves consistently high performance, without a-priori assumptions on the distributions from which the training instances are drawn. It involves a training set of both positive and negative cases. A new instance is classified by calculating the distance to the nearest training case. The KNN classifier extends this idea by taking the k nearest points and assigning the sign of the majority. It is common to select a value for k that is both a small and odd number in order to break ties (typically 1, 3 or 5). Larger k values help reduce the effects of noisy points in the training data set, and the choice of k is often done by means of performing cross-validation. [41]

Evaluating machine learning models

Accuracy: Accuracy (equation 2.1), measures the proportion of the files classified correctly, to the total number of files. Accuracy, however omits a detailed analysis such as the number of correct labels of different classes [72] and in this respect researchers in order to evaluate the model also use the F1 Score as well as precision and recall which are described below.

$$Acc = \frac{true\ positives + true\ negatives}{true(positives + negatives) + false(positives + negatives)} \tag{2.1}$$

Precision: Precision (equation 2.2) measures the proportion of files that were correctly classified as faulty over the total number of files classified as either faulty or non-faulty. In other words, Precision or Confidence (as it is called in Data Mining) denotes the proportion of Predicted cases that are indeed real faulty files [62]. This is a measure of how good a prediction model is at identifying actual faulty files.

$$Precision = \frac{true\ positives}{true\ positives + false\ positives} \tag{2.2}$$

Recall: Recall (equation 2.3) measures the proportion of faulty files which are correctly identified as faulty over the total number of faulty files available. Recall or Sensitivity (as it is called in Psychology) is the proportion of real faulty files that are correctly predicted as faulty files[62].

$$Recall = \frac{true\ positives}{true\ positives + false\ negatives} \tag{2.3}$$

F_1**Score:** The F_1Score is computed by taking the (weighted) harmonic average of precision and recall as shown in equation (equation 2.4).

$$F_1Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{2.4}$$

Confusion Matrix: Table 2.1 depicts a simple cross-tabulation of the mapped class labels against those observed in the ground or reference data for a sample of cases at specified locations. The matrix provides a visual foundation for accuracy assessment (Campbell, 1996; Canters, 1997), and provides the basis on which to both describe classification accuracy and characterize errors, which may help refine the classification or estimates derived from it [31].

		Prediction outcome		
		p	**n**	**total**
actual value	**p′**	True Positive	False Negative	P′
	n′	False Positive	True Negative	N′
	total	P	N	

Table 2.1: Confusion Matrix

2.1.3 Technical Debt

The *technical debt* metaphor was first defined by Ward Cunningham in 1992, where most cited words were ‘not quite right code’ [26]. The definition followed many extensions referring to how the software development teams chose to delay certain maintenance tasks in favor of quick and easy fixes while running the risk causing problems in the future. Therefore, the team opts for an easy and quick way to implement a feature or a fix to a bug in the shortest possible time but with greater chances of negative impact in the long run.
Technical debt refers to the debt incurred by any software item that is inappropriate or deviates from standards and which is included in the system due to urgent fixes or lack of time to properly design the system or lack of time to re-factor it. For example: missing or inadequate documentation, not restructuring or re-factoring complex code, not executing the planned test cases, known defects or bugs that are not yet fixed, etc. Below we provide a list of different types of technical debt.

Architecture Debt

This type of debt refers to the type of issues related to the project architecture [46, 76].
Example: Lack of modular components that affect functional or non functional requirements such as poor performance, robustness, etc.
Identification: The way to identify this type of technical debt is through structural analysis, analysis of component dependencies at the architectural level, and identification of modularity violations [9].

Build Debt

This type of debt refers to the build issues that consume more time and computational resources for completing a build process. This type of debt can be amplified by the use of redundant external libraries or files [51].

Example: Use of too many external libraries could result in build debt.
Identification: The way to identify this type of technical debt is by performing a dependency analysis between the different modules, and the use of dependencies due to libraries or the use of externally linked modules [9].

Code Debt

This type of technical debt is found in the source code of projects that do not follow good coding practices. This practice generally makes it difficult to maintain and in worst cases could lead to rewrite of the entire project or module. Code debt could be handled using static code analysis tools [76, 87].
Example: Using wrong naming conventions in variables or using code cloning.
Identification: The way to identify this type of technical debt is by searching for duplicated code, slow or complex algorithms, code written in a way that violates standards, and by analyzing code metrics [9].

Defect Debt

This type of debt is incurred when a bug is logged in the bug tracking system of the project, generally by the testing team but is deferred due to other high priorities or limited resources available to fix the bug [71].
Example: Defer bug fixes to next cycle in order to meet the current project release deadline.
Identification: The way to identify this type of technical debt is by analyzing bug tracking systems to locate uncorrected known defects [9].

Design Debt

This type of debt is incurred when a project ignores the principles of object oriented design thereby resulting in either large classes (God Classes) or tightly coupled classes [76, 87].
Example: Frequently used code could be generalized using parameters instead this block of code is duplicated across the entire project.

Identification:The way to identify this type of technical debt is by analyzing the system for unusual code metrics patterns, code smells, dispersed coupling, duplicated code, god classes, intensive coupling, and issues in the software design [9].

Documentation Debt

This type of debt is incurred in a project when there is missing, incomplete or inadequate documentation. [76, 66]

Example: A complex, large function with no reference to a functional or non-functional requirement, or a function that does not document its API or functionality to an adequate level.

Identification: The way to identify this type of technical debt is by analyzing system documentation, incomplete design specifications, insufficient comments in code and outdated documentation [9].

Test Debt

This type of debt is incurred in a project when certain planned test activities that affect the testing quality are ignored [76, 66].

Example: Skipping certain planned test cases, unable to meet the planned test code coverage.

Identification: The way to identify this type of technical debt is to analyze the system for incomplete tests and low test coverage [9].

Requirement Debt

This type of debt is incurred in a project when there is a trade-off as to what requirements the development team needs to implement versus the time to release or versus the project's budget [46].

Example: Partially implemented requirements, incomplete requirements.

Identification: The way to identify this type of technical debt is by examining the requirements backlog list [9].

Infrastructure Debt

This type of debt is incurred when there is an infrastructure issue that could hinder the development activities. [67]
Example: Infrastructure failure.
Identification:The way to identify this type of technical debt is to examine the underlying infrastructure (e.g. the middleware) for suitability towards meeting the system's functional or non-functional requirements.

People Debt

It refers to people issues that could delay or hinder the software development activities. [76, 67]
Example: Few subject matter experts, delay in training.

To sum it all up, *technical debt* refers to the difficulty maintaining and evolving a software component due to quick and not well-thought fixes, applied throughout a component's operational life. Examples of technical debt include missing or inadequate documentation, non-critical errors left uncorrected, skipping re-factoring opportunities etc. As there are many different types of technical debt including *architectural debt*[46, 76], *build debt* [51], *code*

debt [76, 87], *defect debt*[71], *design debt* [76, 87], *documentation debt* [76, 66], and *test debt* [76, 66], here we only focus on *code debt* and *defect debt*.

2.1.4 GitHub

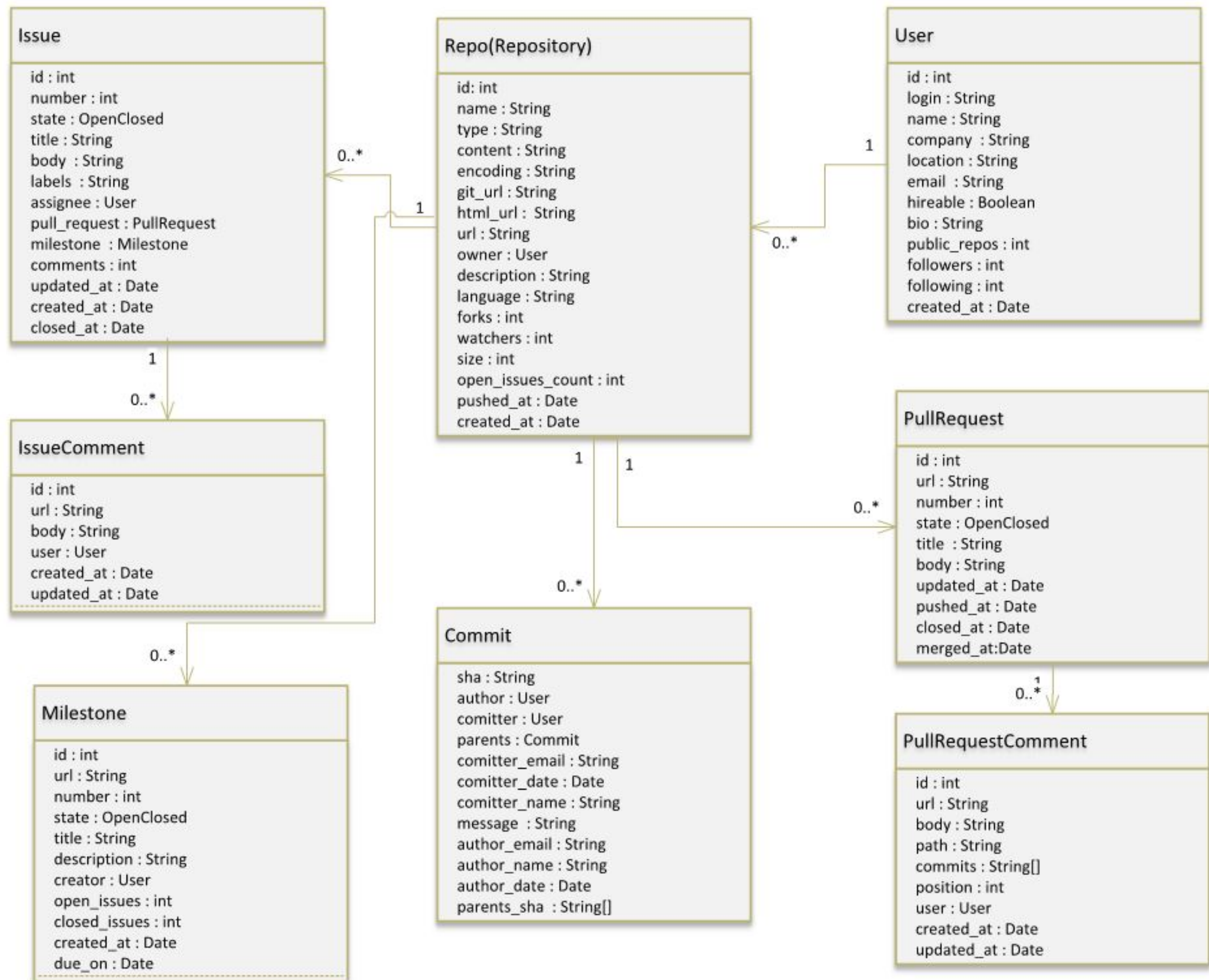


Figure 2.1: Overview of Github's [1] API Data Model

GitHub [1] is a web-based hosting service for version control using Git and it is the preferred repository hosting service for many open source projects. Interestingly, GitHub provides an extensive REST API (5000 calls/hour for authenticated users), thereby, making it attractive for researchers.

Figure 2.1 depicts an overview of GitHub's API Data Model. The entities are *User*, *Repo*, *Issue*, *Commit*, *PullRequest*, *PullRequestComment*, *IssueComment* and *Milestone*. All of the entities have one to many relationship. For example, a *Repo* may have zero or more number

of *Commit* or *Issue* or *PullRequest*. The *id* is a common attribute for all the entities which is a unique identifier except for *Commit* which has *sha* attribute as its unique attribute. When a *User* commits a file, a unique ID which is known as ”sha” or ”hash” is created.

The *User* entity contains user information such as the *login*, *name*, *email*, count of public repositories (*public_repos*), etc. The *Repo* entity contains *type* attribute whose value could be a ’file’, ’user’, ’symlink’ or ’submodule’ and the *content* is set respectively. For example, if the value of *type* attribute is ’file’ then *content* attribute stores its respective data (usually by base64 *encoding*). The *Issue* entity contains information related to all the issues encountered with the project. It has a reference to *PullRequest* which is used to let *User* know about the changes made to repository. *PullRequest* is associated with zero or more *PullRequestComment* which contains the attribute *commits*.

2.2 Related Work

2.2.1 Software Bug Prediction using Machine Learning

There is a variety of machine learning methods that have been proposed for addressing the software bug prediction problem. These methods include decision trees [44], neural networks [91, 75], Naive Bayes [50, 78, 37], support vector machines [33], Bayesian networks [58] and Random Forests [19].

R Malhotra in [48], conducted a systematic literature review of the software bug prediction techniques in 64 primary studies and concluded that the most used machine learning techniques are:

1. Decision trees (DT)
2. Bayesian learners (BL)
3. Ensemble learners (EL)
4. Neural networks (NN)
5. Support vector machines (SVM)
6. Rule based learning (RBL)
7. Evolutionary algorithms (EA)
8. Miscellaneous Approaches

Also, in [48], R Malhotra identifies that the most commonly used metrics for predicting fault proneness are divided into four categories as follows:

1. Procedural metrics: These approaches use metrics that include static code metrics defined by Halstead and McCabe, as well as the LOC related metrics.

2. Object-oriented metrics: These approaches use metrics that measure various object oriented software attributes such as cohesion, coupling and inheritance for an object- oriented class. In addition to object oriented metrics, these approaches also use LOC type of metrics adjusted for the object oriented nature of the code (e.g. taking into account inheritance).

3. Hybrid metrics: These approaches use both object- oriented metrics and procedural metrics to predict fault proneness.

4. Miscellaneous metrics: Metrics such as requirement metrics, change metrics, network metrics extracted from the dependency graph, churn metrics, defect slip through metrics, process metrics, age of file, size, changes and defects in previous versions, elementary design evolution metrics and other miscellaneous metrics that can not be grouped as either procedural or object-oriented metrics, are tagged under Miscellaneous metrics [48].

The object-oriented metrics that are shown to be correlated to fault proneness are CBO (coupling between objects), RFC (response for a class) and LOC [48]. The other metrics that are correlated are WMC, NPM and LCOM. For procedural metrics, the studies do not yield a conclusive result.The results obtained from the primary selected studies indicate that the NOC (number of children) and DIT (depth of inheritance tree) as not useful metrics [48].

R Malhotra [48], identified also the most commonly used metrics to measure performance of the models as well as, the mean performance of these models. More specifically, the study identified that the most commonly used performance metric is Recall which is closely followed by Accuracy, Precision, AUC measures, Specificity, Pf, and Fmeasures. Some less commonly metrics are grouped in the miscellaneous category namely Hmeasure, Precision-recall curve and error rate. The results are provided in the 64 primary studies had the mean values of accuracy ranging from 0.75 to 0.85.

The work in [48] examined also the prediction capability of the machine learning techniques for classifying a file as buggy or not buggy. The machine learning models for estimating software fault proneness outperform the traditional probability models. Based on the results obtained from the systematic review, they also conclude that the machine learning techniques have the ability for predicting software defects and can be used by software practitioners and researchers. However, the machine learning techniques in software bug prediction is still in its initial stages and very limited, and more work should be carried out [48, 80].

HK Dam et al. in [27], performed software defect prediction on two datasets, one from open source projects provided by Samsung and the other, from the public PROMISE repository with a total of 10 Java projects. HK Dam discusses the implementation of a deep learning, tree-structured Long Short Term Memory network which maps with the Abstract Syntax Tree representation of the source code. The entire process is based on three steps.

The first step is to parse a source code file into an Abstract Syntax Tree(AST). The second step is to map the AST nodes into continuous-valued vectors called embeddings and then input these embeddings to a tree-based network of LSTMs to get a vector representation of the whole AST. The third step is to input this vector into a traditional classifier (e.g. Logistic Regression or Random Forests) to predict defect outcomes.

The results of the model reported in [27] for *within-project* prediction achieved a very good recall of 0.86 (averaging across 16 cases). However, this approach has lower precision

and average AUC of 0.6. For cross-project prediction, their approach did achieve a very high recall, with an average of 0.8 across 22 cases. The average F-measure was reported 0.5 but the approach suffered from low precision.

Xinli Yang et al. in [85], performed defect prediction experiments on 6 large-scale software projects from different communities, i.e., Bugzilla, Columba, JDT, Platform, Mozilla, and PostgreSQL. The authors present the overall framework of their proposed approach called 'Deeper' which is summarized as follows. The procedure consists of two parts: a model building phase and a prediction phase. In the model building phase, the goal is to build a classifier by using deep learning and machine learning techniques from historical code changes. In the prediction phase, this classifier would be used to predict if an unknown change would be buggy or clean. In [85], Xinli Yang et al. used 14 basic features such as the number of modified subsystems, directories, files, the number of lines added, deleted, total lines of code and developer's experience to train their model. The results obtained using the 'Deeper' model indicate that the best the model could do in terms of precision, recall and accuracy are 0.55, 0.72 and 0.62 respectively.

Wójcicki et al. in [83], want to verify whether the type of approach used in former fault prediction studies can be applied to Python and opted for using a Naïve Bayes classifier for fault prediction. The model used McCabe's cyclomatic complexity measure, counters of operators and operands (Halstead metrics) as features. The results achieved recall up to 0.64 with a false positive rate of 0.23. The mean recall and mean false positive rate were reported as 0.53 and 0.24 respectively.

Venkata et al. in [22], compared different machine learning models for identifying faulty software modules and they found that there is no particular learning technique that performs the best for all the data sets. In that study, the metrics used were McCabe, Halstead, line count, operator and branch count. The models used were Decision Trees, Naïve-Bayes, Logistic Regression, Nearest Neighbor, 1-Rule and Neural Networks. Venkata et al. in their results show that the "size" related and "complexity" related metrics are not sufficient attributes for accurate prediction, and they need to include dependencies between these metrics to improve the prediction models.

Wang and Yao in [81], aimed to find bugs without decreasing the overall performance of the model. In their study, the machine learning algorithms used were Naive Bayes (NB), and Random Forest (RF). The dataset used for training the model is taken from public PROMISE repository. In this process, they find that imbalanced distribution between classes in bug prediction is the root cause of its learning difficulty. Likewise, in our thesis, we noted the issue and used re-sampling as described in detail in the section 3.4.2.

Zimmermann et al. in [93], propose an approach to predict bugs on cross-language systems. The work examined a large number of such systems and concluded that only 3.4% of the systems had a precision and recall prediction levels above 75% . The authors also tested the influence of several factors on the success of cross-language prediction and concluded there was no single factor that led to such successful predictions. The authors used decision trees to train the model and to estimate precision, recall, and accuracy before attempting a prediction across systems.

2.2.2 Software Metrics

Object Oriented Metrics

DL Gupta et al. in [34] investigate and assess the relationship between different object-oriented metrics and defect prediction capability by computing the accuracy of their proposed model on different datasets obtained from different repositories for a given project. A total of 12 experiments were carried out on different dataset sources, using 14 different classifiers for each experiment. The metrics used to train the model are WMC, DIT, NOC, CBO, RFC, LCOM, CA, CE, LOC and LCOM3, NPM, DAM, MOA, MFA and CAM. The classifiers used are Naive Bayes, LivSVM (Support Vector Machine), Logistic Regression, Multi-Layer Perceptron, SGD (Stochastic Gradient Descent), SMO (Sequential Minimal Optimization), Voted Perceptron, Attribute Selected Classifier, Classification Via Regression, Logit Boost, Tree Decision Stamp, Random forest, Random Tree and REP (Reduce Error Pruning) Tree. On an average, 76% accuracy is achieved at testing (prediction) stage when training is done on all the datasets. It is concluded from all the experiments, that some datasets have similar prediction accuracy result, while some give different results for the same projects on other datasets. Similar conditions occur when training and testing are both done on all the datasets and this type of variation in the accuracy at prediction (testing) stage may be due to different softwares (datasets) used for the experiments.

K El Emam et al. in [29], based on the previous work on predicting faulty modules using object oriented design metrics, such as DIT, NOC and the Briand et al. coupling metrics (like CBO and RFC). To perform validation of object-oriented design metrics, a commercial Java system is used. The objective of the validation is to determine which of these metrics are associated with fault-proneness. The metrics used in the validation are DIT, NOC, ancestor-based coupling metrics and descendant-based coupling metrics. The results indicate that inheritance depth and coupling metric were strongly associated with fault-proneness. The prediction model with these two metrics has promising accuracy but only one system was considered for the experiment.

M Cartwright et al. in [18], investigated an industrial object-oriented module using empirical methods. The module has a size of 133 KLOC, is written in C++ and is part of a larger real-time European telecommunication product which comprises several million lines of code(LOC) and has been evolving over the past 10 years. The metrics used to experiment were LOC, the number of all read accesses by a class, the number of all write accesses by a class, depth of inheritance (DIT), number of child classes (NOC) and the number of defects of each class. The experiment that used the Shlaer-Mellor method [69] concluded that object-oriented constructs such as inheritance and polymorphism are not really useful to predict defects. According to the authors in [18], the classes involved in inheritance structures were three times more defect prone than the classes that were not involved in inheritance structures. On the other hand, these results may be a consequence of the development method used, or the fact that the C++ language does not enforce an object-oriented approach, or the fact that this was the project team's first experience of object oriented development.

Nagappan in [55], aims to find the best code metric to predict bugs. The conclusion of this work is that complexity metrics can successfully predict post-release defects, but there is no single set of metrics that is applicable to all systems.

Menzies et al. in [50] used static code metrics such as SLOC and CCN for defect prediction. According to Menzies et al, they observed that the prediction performance was not affected by the choice of static code metrics but by how the chosen metrics are used. Therefore, the choice of selecting and training a machine learning algorithm plays more an important role than the metrics used for training the machine learning algorithm. The authors, measure performance by using Probability Detection (PD) and Probability of False alarms (PF). Using a Naïve Bayes algorithm with feature selection, they gained a PD score of 71% with PF at 25%.

Zhang et al. in [89] commented on them using PD and PF performance measures and proposed the use of precision and recall instead. Zhang [88] investigated if is a co-relation between LOC and defects on publicly available datasets. Zhang et al. argued that LOC when combined with machine learning classification techniques, could be a useful indicator of software quality. The results are definitely interesting because LOC is one of the simplest and easy to collect software metric.

Repository Metrics

Repository metrics are also known as change or process metrics. Repository metrics are metrics that are based on historic changes made on source code over a period of time. These metrics can be extracted from version control systems. A few examples of repository metrics are the number of additions and deletions of lines from the the source code, total number of altered lines of code, the number of authors of a file, the number of commits a file, etc.

AE Hassan in [36], discusses how frequent source code "commits" in the repository, negatively affect the quality of the software system, meaning that the more changes incurred to a file, the higher the chance that the file will contain critical errors. Furthermore, Hassan in [36] presents a model which can be used to quantify the overall system complexity using historical code-change data, instead of plain source code features.

B Caglayan et al. in [17] discuss about the merits of using repository metrics for bug prediction and have concluded that repository metrics provide a better insight to the software product and lower the probability of false alarms. Moreover, the authors in [17] concluded that static code attributes provide limited information content. Software modules represented in terms of static code attributes may overlook some important aspects of software, including the type of application domain; the level of skills of the individual programmers involved in system development; contractor development practices; variation in measurement practices; and the validation of measurements and instruments used to collect the data [17]. There have been a few approaches that use repository data to predict defects but they did not consider the number of developers. They found that a high number of programmers coupled with commits to the same file, as well as a high temporal concentration of commits, were associated with class level quality problems, and in particular size and design guidelines violations [17].

Moser et al. in [52] compared the efficiency of using repository metrics for defect prediction to static code metrics. Their conclusion was that process data contained more information on the distribution of defects than the source code metrics. Their explanation is that the source code metrics concern the human understanding of the code. For example, large files of source code do not necessarily mean that the file is fault prone.

Nagappan et al. in [54] used repository metrics with respect to code churn, such as number of commits of a file, number of additions or deletion of line of code code over a specified period

of time. Nagappan et al. concluded that the use of relative code churn metrics better predict the defect per source file than other metrics, and that code churn can be used to distinguish between defective and non-defective files. By relative churn measures the author takes into consideration the normalized values of the various measures obtained during the churn process. For example, a few of the normalization parameters are total lines of code, file churn, file count etc.

Ostrand et al. in [60] used repository metrics explaining whether a file was new and whether it was modified or not. The authors used these metrics in addition to other metrics and found that 20 percent of the files, identified by the prediction model as most fault prone, contained on average 83 percent of the faults.

Test Metrics

Test code metrics may consist of the same set of code metrics as described earlier, but their only difference is now that they are applied on the source code of the test classes. Adding to this, Nagappan et al. in [56] introduced a test metric suite called Software Testing and Reliability Early Warning metric suite (STREW) in order to find defects in software programs. In STREW, Nagappan et al. considered nine metrics belonging to three different families namely Test quantification metrics; Complexity and object-oriented metrics and; Size adjustment metric. Nagappan et al. conclude that the metrics used in STREW provide at an early stage of development an estimate of the quality of the software as well as the identification of fault-prone modules [56]. Further studies conducted by the authors indicated that the STREW metrics suite for Java systems (also know as 'STREWJ') can effectively predict software quality.

Dependency Metrics

In the design phase of the software development life-cycle, the division of the workload of different components can be decided. However, this division of tasks can influence the quality of the software. Therefore, tracking dependencies between source code files could be a possible predictor of fault-prone components. Schröter et al. in [65] proposed a new approach for detecting source files and packages for fault prone code. The authors collected import statements for each source code file or package, and compared these imports to component failures. The authors concluded that the dependent components, determines the likelihood of defects. The results show that the collection of imports at package level results in a better prediction than at file level. However, the results on the file level still work better than random assumptions.

Hybrid metrics

Previous studies [17, 52] combined metrics from several metric families to achieve better prediction performance. Caglayan et al. in [17] compiled three different sets of metrics. In the first set, the authors used only static code metrics. In the second set, they used only repository metrics, while the third set, was a combination of the first two. The results of this study showed that while the prediction accuracy was not improved, the false positive rate was decreased when the combined set was used. Moser et al. [52] built a static code set, a repository metric set, and

a static and repository set combined. They concluded that the combined set was carried out just as well as the repository set, indicating that static code metrics are not worth collecting.

Technical Debt Metrics

N Zazworka et al. in [86] compared four approaches for technical debt identification. A number of source code analysis techniques and tools have been proposed to potentially identify the code debt accumulated in a system. N Zazworka et al. also investigated whether technical debt, as identified by the source code analysis techniques, correlates with *interest indicators* in the form of ***increased defect-proneness and change-proneness***. The selected four techniques for the analysis are identification of code smells, application of static analysis, identification of grime buildup (non design pattern related code which is included in the code that implements a design pattern), and modularity violations. The four techniques are applied to 13 versions of the Apache Hadoop open source software project to calculate *technical debt*. The indicators used in each of the techniques are: 1) Grime - Presence of Grime, Absence of Design Pattern; 2) Code smells - class level code smells such as God class and method level code smells such as coupling; 3) Source code analysis techniques - Finding bugs by priority (High, Medium, Low) and by category (Performance, Security, Experimental) and, 4) Modularity violations - Is there any presence of modularity violation?

N Zazworka et al., in his study, had the following observations:

- The value of technical debt indicators (Modularity violations, Grime, Code Smells and Source code analysis techniques) increases together with the size of the project.
- As for correlations between technical debt indicators and interest indicators (defect proneness and change-proneness), dispersed coupling points to classes that are more defect prone.
- Lastly, defect-prone classes also tend to be change-prone and vice versa.

J Xuan et al. in [84], proposes the concept of 'debt-prone bugs' to model the technical debt in software maintenance. Debt-prone bugs are considered to be the debt incurred by maintenance operations, and produced by an incomplete or immature process of bug fixing and can add risks to software quality. Three types of debt-prone bugs, namely tag bugs, reopened bugs, and duplicate bugs were considered for examination. Three attributes were extracted for each of the types of debt-prone bugs. These are the number of bugs, the frequency of debt-proneness and the time of fixing bugs. To investigate the correlation between debt-prone bugs and product quality, prediction models were constructed based on historical information to predict the time related to fixing bugs.J Xuan's et al. contributions can be summarized as follows.

First, the authors propose the concept of debt-prone bugs that extends the existing technical debt in software maintenance.Second, they identify three types of software debt-prone bugs, namely tag bugs, reopened bugs, and duplicate bugs. Tag Bugs relate to TODO or FIXME tags which are inserted by developers to prompt unfinished work. However, some of the TODO tags may be forgotten or even become bugs because of the accumulation of TODO or FIXME tags in the code. Reopened Bugs relate to the situation where a bug may be solved by developers in

bug tracking systems, but reopened later by another developer who realized that the bug was not fixed properly. The study showed that the time associated to fix reopened bugs is longer than the time associated for other bugs. These bugs are labeled as second type of debt-prone bugs. A Duplicate Bug is a new bug that has the same root cause in the bug tracking system as an existing bug. Ideally, duplicate bugs can be avoided if a developer is familiar with all existing and related bugs. However, in practice, developers can not check all existing bugs to determine whether a new bug matches an existing one. Since duplicate bugs are caused by the inadequate examination, these bugs are labeled as the third type of debt-prone bugs.

Finally, the authors in [84] conducted a case study to examine the correlation between debt-prone bugs and software quality in the Mozilla project. Their results indicated that the time related to fixing duplicate bugs can be seen as the strongest correlation factor to the average time of bug fixing. One possible reason for this, is that the duplicate bug ratio is greater than the other two types of debt-prone bugs. The Mozilla project experiment indicated also that the debt-prone bugs can help monitor and improve the quality of the software system.

2.2.3 Results Comparison

The average F1 Score, Precision, Recall and Accuracy for *RQ1* using code metrics are 0.61, 0.64, 0.59, 0.72 respectively and using repository metrics are 0.72, 0.77, 0.75 and 0.82 respectively. For *RQ2*, the average F1 Score, Precision, Recall and Accuracy for Java projects are 0.60, 0.59, 0.68, 0.84 and for Python projects are 0.73, 0.74, 0.73 and 0.73 respectively. HK Dam et al. in [27], performed software defect prediction with a total of 10 Java projects using AST as features. The results reported in [27] for *within-project* prediction achieved a very good recall of 0.86 (averaging across 16 cases). However, their approach has lower precision and average AUC score of 0.6. For *cross-project prediction*, their approach did achieve a recall, with an average of 0.8 across 22 cases. The average F-measure was reported 0.5 but the approach suffered from low precision. Xinli Yang et al. in [85], performed defect prediction experiments on 6 large-scale software projects from different communities, i.e., Bugzilla, Columba, JDT, Platform, Mozilla, and PostgreSQL using 14 code and repository metrics. The best their model could do in terms of precision, recall and accuracy are 0.55, 0.72 and 0.62 respectively. Wójcicki et al. in [83], applied to Python projects and used McCabe's cyclomatic complexity measure, counters of operators and operands (Halstead metrics) as features. Their results achieved recall scores up to 0.64 with a false positive rate of 0.23. The mean recall and mean false positive rate were reported as 0.53 and 0.24 respectively. Menzies et al. in [50] used static code metrics such as SLOC and CCN for defect prediction and reported a Probability Detection (PD) score of 71% with False alarms (PF) at 25%. Zimmermann et al. in [93], predicts bugs on cross-language systems using combination of code and repository metrics and reported a precision and recall prediction levels above 75% . R Malhotra in [48], identified the results provided in the 64 primary studies had the mean values of accuracy ranging from 0.75 to 0.85. The above results indicate that this is still an open area in Experimental Software Engineering and there is still work to be done for identifying the appropriate features as well as the appropriate models to achieve highly accurate software fault proneness prediction models. Furthermore, the results indicate that the approach presented in this thesis is able to perform predictions of software fault-proneness by utilizing features that can be easily extracted from source code and other DevOps repositories and at the same time being programming language agnostic.

Chapter 3

Fault-proneness Prediction

In this chapter, we present a framework to utilize code and repository metrics in a machine learning environment in order to predict fault-proneness at the file level. Our study was conducted along three dimensions. The first dimension has to do with the selection of appropriate systems and appropriate code and repository metrics to use. The second dimension relates to the selection of the appropriate machine learning techniques. The third dimension deals with the training and evaluations of the models in several open source systems.

3.1 System Selection

3.1.1 Criteria

In [43], the authors document the results of an empirical study aimed at understanding the characteristics of the repositories and of users on GitHub. The empirical study talks in depth about mining GitHub for research purposes and how one should take various potential perils into consideration. The authors in their study listed 13 perils that were discovered. Our selection of projects from GitHub was based upon these perils. More specifically, our project selection was based on the following related criteria.

C1: *A repository is not A project:* The work in [43] concluded that a project is typically part of a network of repositories: at least one of them will be designated as central, where code is expected to flow to, and where the latest version of the code is to be found. Keeping this into consideration, we carefully selected projects that were not a part of a network of projects.

C2: *Most projects have low activity or are inactive:* Most of the projects have very low activity or are inactive [43]. Taking this into consideration, we only selected projects that had high activity in development, discussions and bug reports.

C3: *Many repositories pertain to personal projects and do not relate to a major software application:* Most of the projects stored in repositories relate to toy projects undertaken by developers [43]. Taking this into consideration, we selected project repositories that relate to open source software applications which are widely used.

C4: *Many active projects do not use GitHub exclusively and few projects use pull requests:* Most of the projects do not use GitHub exclusively and only a few projects use pull requests [43]. It is true that very few projects use pull requests and therefore, it was very challenging to find large open-source projects that use pull requests. Of the projects we used for our study, all of them indicate a very high GitHub activity. However, we are unable to certify that GitHub was exclusively used in all of these projects.

C5: *Only the user's public activity is visible and GitHub's API does not expose all data:* The user's public activity is visible and the API does not expose all the data [43]. For our experiment, we selected projects for which the data we needed were available.

3.1.2 Method

Figure 3.1 depicts the overall process of our framework. The first step relates to the selection of systems to be considered. We have selected a total of ten projects comprising five Java and five Python projects based on the criteria C1-C5 listed above. The second step relates to applying custom made programs to retrieve information pertaining to these projects from the corresponding GitHub repositories. The third step relates to modeling and reconciling the extracted information into a form that can be used by a machine learning framework. The fourth step relates to selecting ten different popular machine learning techniques (SVC, Decision Tree, AdaBoost, RandomForest, ExtraTrees, GradientBoosting, MLP, KNN, Logistic Regression and LDA) so that models can be trained to answer question RQ1. Out of these ten models, we selected the five (SVC, DecisionTree, AdaBoost, RandomForest and KNN) top most performing models, based on their cross validation score (please see detailed report in Section B.1), in order to use these five models for answering the research questions RQ2 - RQ4. To answer the research question RQ5, this step was repeated but considering only advanced source code metrics instead of our standard selected set of metrics. The fifth step relates to obtaining the best performing models from questions RQ2 and RQ3 and applying it to two C++ projects obtained from KDE, in a quest to evaluate the applicability of our approach to systems written in a different language than the one the models were trained on.

3.1.3 Selected Systems

To conduct our study, we considered metrics obtained from both the source code of the system, and from GitHub repositories related to the DevOps tools. In order to obtain these metrics and information, we have developed a Java client program to automatically connect to various repositories, metrics collection tools, as well as programs to aggregate, reconcile, and model all the obtained information into one data model, so that it can be used for training and testing purposes.

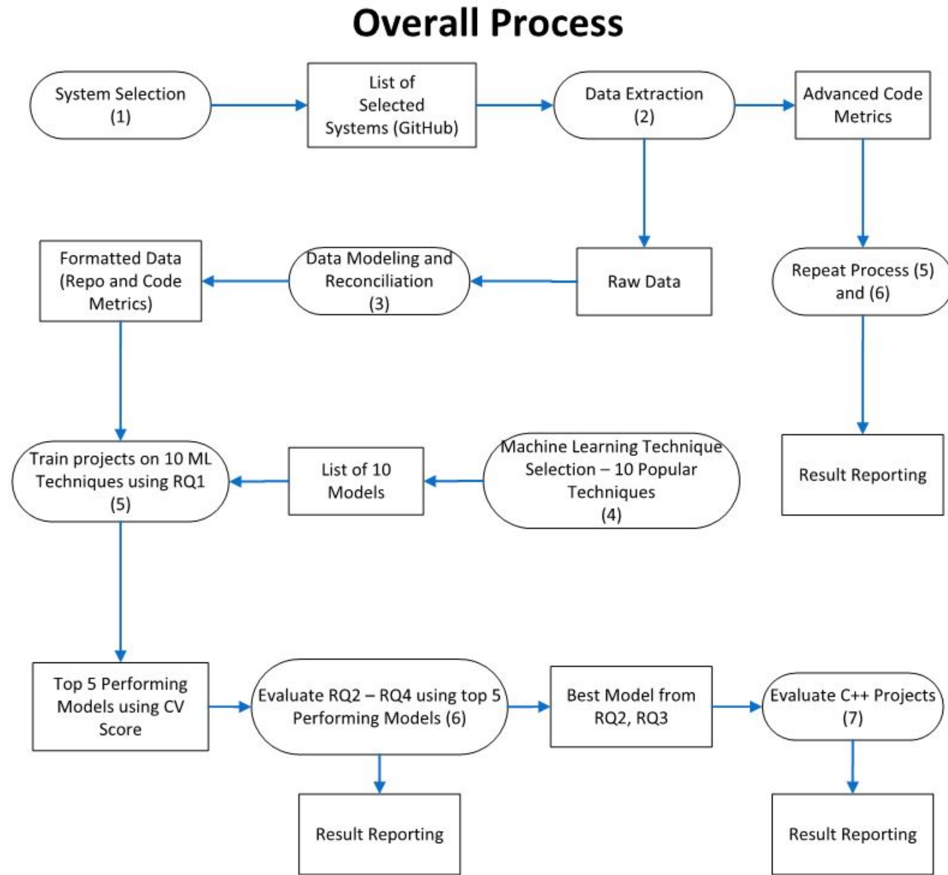


Figure 3.1: Overview of project selection, research questions, experiment and validation

Table 3.1: Systems used in the experiment.

Project Name	Language	Files	Lines of Code
spring-boot	Java	4225	249,226
deeplearning4j	Java	1845	498,447
elasticsearch	Java	8460	1,087,149
eclipse-che	Java	9159	704,659
RxJava	Java	1604	273,053
youtube-dl	Python	824	115,037
ipython	Python	215	46,626
sci-kit	Python	233	158,207
scrapy	Python	240	27,439
keras	Python	82	47,253

The selection criteria for the software applications to consider for our study were described in detail in Section 3.1.1. In addition to the selection process in Section 3.1.1, we also looked into the project size in terms of the number of files and lines of code. Larger systems were preferred over smaller ones in order to make the results more accurate and generalizable. We also looked at the availability of accurate information about software bugs and resolutions, so that the training and testing phases can be adequately supported. Lastly, we considered the overall complexity of the application as more complex applications entail a greater number of dependencies and opportunities for training. The software applications we considered in our study are listed in the Table 3.1 comprising of five Java and five Python systems, for which there is source code and repository related information available on GitHub (github.com) [1].

3.2 Metric Selection

3.2.1 Repository metrics

Fractal: This feature represents the percentage of code written by the original author that remains in the system after a series of commits over a period of time.

The fractal data provide another investigative tool to reveal files which undertook significant change compared to the original code. The visualization of fractal figures provides a view of how the programming effort was shared i.e. how fragmented the developer effort is for each module in a system[77]. An example is shown in Figure 3.2 where in file car.cpp, the value is 1 (i.e. all the code is the code of the original developer), while in file ClassShip.java the value is 0.25 indicating that only 25% of the original code remains.

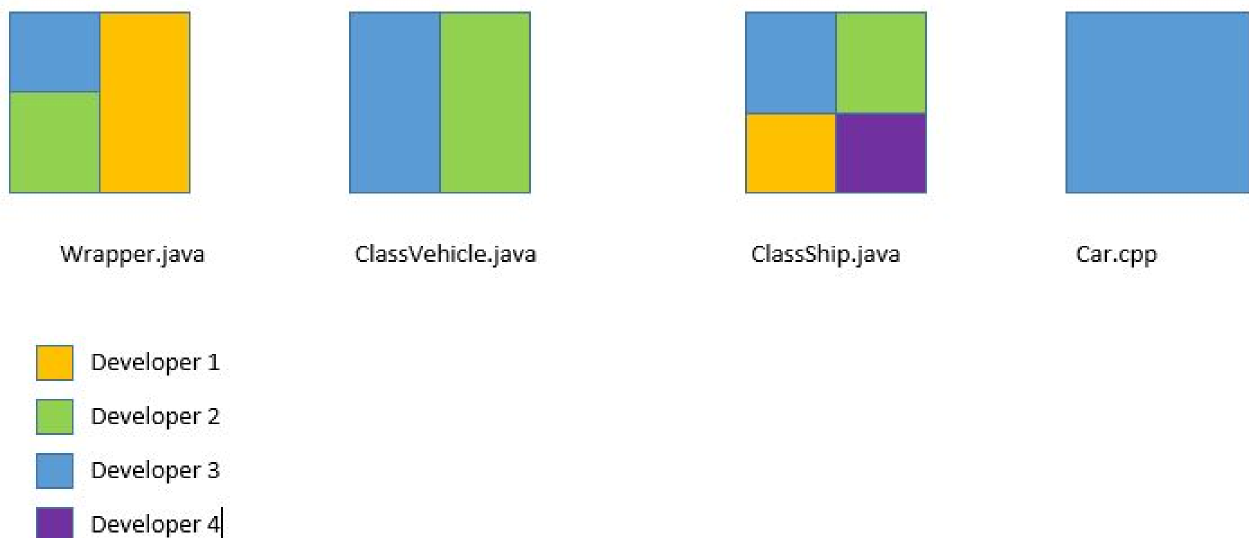


Figure 3.2: Fractal visualization example [77]

Sum of Coupling (SOC): *Sum of coupling* provides information on how many times in total each module has been coupled to another one in a commit operation. For example, as depicted in Figure 3.3 the module *app.clj* is modified together with both module *core.clj* and module *project.clj* in commit #1, and module *core.clj* in commit #2. Therefore, its *sum of coupling* value is three. The premise is that the module that changes most frequently together with others, must be important and should a be good starting point for an investigation. There are different reasons for modules to be coupled. Some couples, such as a unit and its unit test, are valid. So modules with the highest degree of coupling may not be the most interesting to us. Instead, we want modules that are architecturally significant. A sum of coupling analysis helps locating those modules.

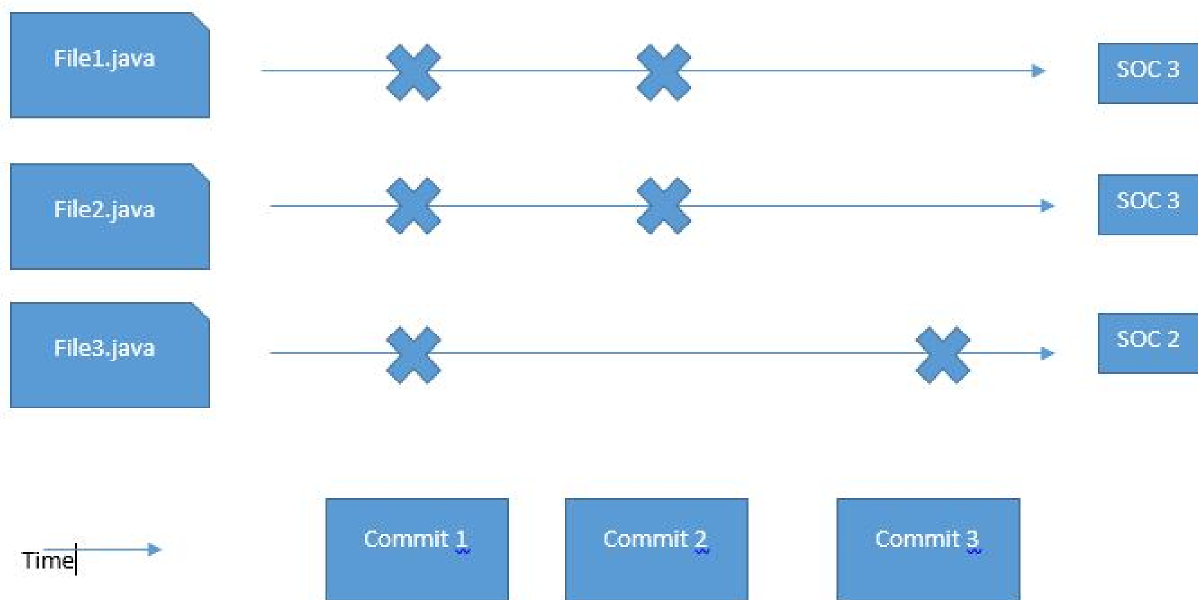


Figure 3.3: Sum of Coupling visualization example [77]

Age: This metric represents the age of the file in months.

Authors: This metric represents the total number of authors who have worked on the file.

3.2.2 Code metrics

Java code metrics: The code metrics used as features are described below.

McCabe's Cyclomatic Complexity (McCC): Complexity of a method expressed as the number of independent control flow paths in it [5]. It represents a lower bound for the number of possible execution paths in the source code and, at the same time, it acts as an upper bound for the minimum number of test cases needed for achieving full branch test coverage.

Based on McCabe's Cyclomatic Complexity, the complexity of the whole file is expressed as the number of independent control flow paths in it. It is calculated as the sum of the McCabe Cyclomatic Complexity values of the methods found in a file [5].

Comment Lines of Code (CLOC): It represents the number of comment and documentation lines in a source file [5].

Public Documented API (PDA): It represents the number of documented public classes and methods in a file [5].

Public Undocumented API (PUA): It represents the number of undocumented public classes and methods in a file [5].

Logical Lines of Code (LLOC): It represents the number of non-empty and non-comment code lines in a file [5].

Lines of Code (LOC): It represents the number of lines of code in a file [5].

Python code metrics: Similar to Java code metrics (excluding PDA and PUA), Python systems have McCC, CLOC, LLOC, LOC and Number of Statements (NOS) as the features used to train the model.

Number of Statements (NOS): It represents the number of statements in the file [6].

These features were selected using Pearson's correlation coefficient [14] and the tool that was used to plot Pearson's correlation matrix was pandas' correlation function and pyplot [40].

3.3 Data Collection Process

In order to collect data from GitHub, for both repository related information (e.g. commits, bugs, bug resolutions) and code metrics, a Java client program was developed and used in order to query GitHub as shown in Figure 3.4. The Java client program fetches every bug that was recorded for a file and keeps a count of the number of times it was found to contain an error (bug) that produced a failure. Other information that is fetched includes the number of source code commits, as well as the type of operation involved. The client program also connects with tools that compute source code metrics, and automatically populates a data base with all this information. Finally it performs data reconciliation, ensuring that the obtained data pertain to the same system, module and version. The bug data is then individually combined with the extracted repository metrics [8] and code metrics [4] to form two separate data files, one for predicting failures using repository features and another for doing the same using code metric features. The two individual results are then superimposed (intersected) to form one final result.

Below is the pseudo code (please see complete Java program in Section A.1.1) that is used to fetch all the bug reports from GitHub. The program is divided into three parts namely project configuration, bug data extraction, and finally exporting the extracted data to a csv formatted file.

In the first part i.e. project configuration, we connect using OAuth by providing the OAuth token that is acquired by creating an account on GitHub. Then, we specify the project name to connect and extract its details.

In the second part of the program, for every bug that is reported, we look at the resolution and procure the list of files that were responsible for this bug. All of this data, that is, bug issue

id, issue description, issue title and issue state are stored in a wrapper object.

In the third part of the program, all the data that we store in the wrapper class are now written into a csv file in order to combine these results with code and repository metrics. The algorithm for bug extraction is in Section A.1.1 and below is the pseudo code for the same.

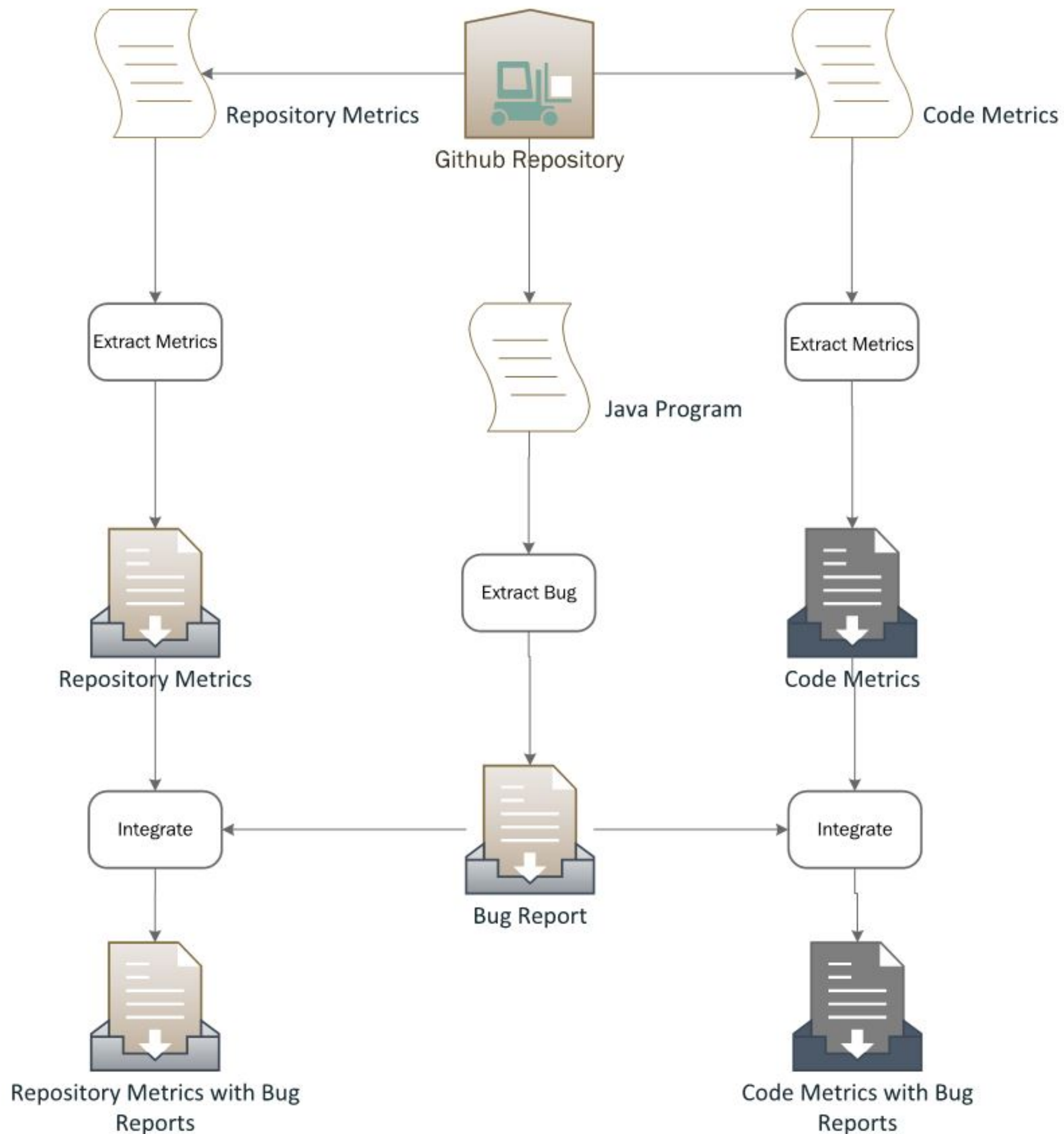


Figure 3.4: Data collection from [1] using metrics tools [4, 8].

Pseudo code for bug extraction from GitHub: The pseudo code for extracting bug information from GitHub is shown in Algorithm 1. The complete Java code for bug extraction is in Section A.1.1.

Algorithm 1 Pseudo code for bug extraction from GitHub

```
1: procedure ISSUESSUMMARY          ▷ This class is used to fetch all bugs from GitHub
2:     Initialize oauthToken,GitHubBuilder,Github
3:     Initialize GHIssue
4:     Initialize GHLabels
5:         ▷ Config: To connect to GitHub, Initialize oauthToken,GitHubBuilder,Github
6:     Initialize GHRepository       ▷ Initialize the Github repository to get the details
7:     for each in GHIssue do
8:         for each in GHLabels do
9:             GET IssueId, Body, Title, BugLabel, state
10:            STORE IssueId, Body, Title, BugLabel, state in LIST
11:        end for
12:    end for
13:    Write LIST to CSV       ▷ Write IssueId, Body, Title, BugLabel, State to csv file.
14: end procedure
```

Now that we have all the list of bug reports, the only way to link it with the files is to fetch all the *commits* for that repository, look for the *issue id* (if it exists as it could be an enhancement or a feature request) and map it with the files responsible for the issue. Below, we provide the pseudo code for the algorithm that extracts the *commits* for the entire repository and maps the issue id with the files responsible for the issue (please see the complete program in Section A.1.2).

Pseudo code for combining extracted bug information with code and repository metrics: The pseudo code for combining extracted bug information from GitHub with code and repository metrics is shown in Algorithm 2,3. The complete Java code is in Section A.1.2.

Algorithm 2 Pseudo code for commit extraction from GitHub

```
1: procedure COMMITSSUMMARY        ▷ This class is used to fetch all commits from GitHub
2:     Initialize oauthToken,GitHubBuilder,Github
3:     Initialize GHCommits
4:     Initialize GHFiles
5:         ▷ Config: To connect to GitHub, Initialize oauthToken,GitHubBuilder,Github
6:     Initialize GHRepository       ▷ Initialize the Github repository to get the details
7:     for each in GHCommits do
8:         for each in GHFiles do
9:             GET Date, Url, Message, Filename
10:            STORE Date, Url, Message, Filename in LIST
11:        end for
12:    end for
13:    Write LIST to CSV                          ▷ Write Date,Filename,Url to csv file.
14: end procedure
```

Algorithm 3 Pseudo code for combining extracted bug information with code and repository metrics

```
1:  procedure ExtractIssueIdfromCommitsSummary ▷ This class is used to map commits and bug reports from GitHub
2:      Load CommitSummary.csv
3:      while csvReader.hasNext do
4:          while commitHasBug do
5:              STORE Date,Filename,issueId,Message,Url in LIST
6:          end while
7:      end while
8:      Write LIST to CSV          ▷ Write Date,issueId,Filename,Message,Url to csv file.
9:      ADD respective metrics     ▷ Add Code or Repository metrics - this step is manual
10: end procedure
```

The bug report for each project is integrated with the corresponding metrics report obtained by the tools reported in [4, 8]. The integrated report was fed to Kaggle [2] in order to train and evaluate the models. The integrated bug reports with repository metrics is shown in Figure 3.5 and with code metrics, it is shown in Figure 3.6.

Path	Bug	soc	frag	authors	age
kioslaves/videodvd/videodvd.cpp	0	13385	0.8	11	1
libk3bdevice/k3bdevicetypes.h	1	10612	0.11	3	1

Figure 3.5: Sample data with repository metrics.

Path	Bug	McCC	CLOC	LLOC	LOC
spring-boot-project/spring-boot/src/test/java/sampleconfig/MyComponentInPackageWithoutDot.java	0	1	0	5	25
spring-boot-project/spring-boot/src/main/java/org/springframework/boot/ApplicationArguments.java	0	1	0	10	75

Figure 3.6: Sample data with code metrics.

3.3.1 Identify files with errors

The information stored in GitHub [1] repositories is organized using predefined schemas. An element important to our work, found in that schema, is the *issues* element, where all the issues for that repository are logged. This schema element includes information about feature enhancements, debugging, request to help, bug reports, etc. The major advantage of choosing large systems is that they are very organized and the bug reports have specific labels, and tracking numbers. The implemented Java client program runs through all the bug reports and collects the data from files which either contain an error or are associated with a symptom.

3.3.2 Collection of repository information and metrics

The entire project code from GitHub [1] is cloned on our local machine and an information collection tool [8] is invoked by the client program in order to gather project related features from the GitHub repository. This information includes bug reports, commits, sum of coupling, fractal value, age of project in months, and the number of authors for each module [77]. The files and repository entries that are associated with a bug report are marked by a '1' flag, while the rest are marked by a '0' flag.

3.3.3 Collection of static code metrics

The entire source code from the system under analysis is also cloned on the local machine. A tool that computes source code metrics [4] for Java and Python systems is invoked by the client program in order to gather source code metrics such as lines of code, logical lines of code and, the number of incoming calls (fan-in), etc. The complete list of features extracted from the tool [4] are presented in [5, 6]. All the features used in the experiment are discussed in detailed in sections 3.2.1, 3.2.2.

The repository related information (e.g. commits, bug reports) and the source code metrics data obtained by the Java client program are then combined to form a reconciled and integrated data base.

3.3.4 Experimentation Framework

The experimentation framework has two parts. The first parts deals with the acquisition of raw data from the GitHub repositories. The second part deals with the training and the compilation of the predictive models. With respect to raw data acquisition we have used a setting of three virtual machines running at Western University Department of Computer Science Research Lab, running using a custom made Java client program for issuing and managing requests to GitHub repositories by. This setting was necessary as GitHub servers limit the number of allowable requests to 5,000/hour. The configuration for each virtual machine was 32GB of RAM, utilizing a 8-core processor running at 2.4 GHz). Upon reception, the data were pre-processed by a) b)

3.4 Building a model

3.4.1 Model

Voting Classifier: From a technical point of view [90], ensemble learning is mainly implemented in two steps: training weak base classifiers and selectively combining the member classifiers into a stronger classifier. Usually, the members in ensemble learning are constructed in two ways. One is to apply a single learning algorithm, and the other is to use different learning algorithms over a dataset [28]. The base classifiers are then combined to form a decision classifier. Generally, to get a good ensemble, the base learners should be as accurate and as diverse as possible. The process of choosing an ensemble of accurate and diverse base learners is the focus of many researchers' work [92].

In our voting classifier, we used different learning algorithms such as SVC, Decision Tree, AdaBoost, Random forest and KNN and these trained algorithms are put together to a voting classifier in order to make the final prediction.

3.4.2 Re-sampling and highly skewed features

Re-sampling imbalanced data: Most of the systems we used for our evaluation study (see Table 3.1) had imbalanced classes (majority of the samples being either positive or negative) and needed re-sampling. In this respect we used the sklearn toolkit [47] to re-sample the training data in order to minimize the resulting bias of the model. The only systems we did not need to re-sample were the ipython, sci-kit and RxJava systems as their classes were close to balance on either side. In Figure 3.7 an example of such class imbalance is depicted that occured for the the springboot system. Such imbalances could lead to our models being biased to that majority class and this is empirically observed in results both before and after re-sampling using repository metrics in Table 4.1a, 4.1b and code metrics in Table 4.2a, 4.2b.

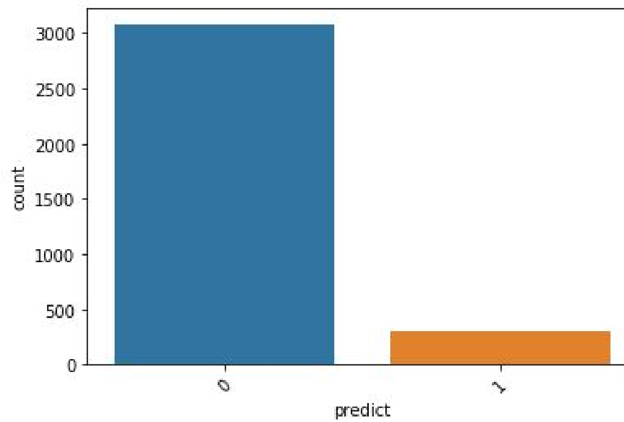


Figure 3.7: Imbalanced classes for springboot project

Box-Cox Transformation: In order to transform highly skewed features and to improve the overall data transformation [59], we have implemented and applied the Box-Cox Transformation [64].

3.4.3 Model evaluation

To evaluate the model, for every obtained result, we calculated the F1 score, precision, recall, accuracy and confusion matrix as shown below.

Accuracy: Accuracy (3.1), measures the proportion of the files classified correctly, to the total number of files.

$$Acc = \frac{true\ positives + true\ negatives}{true(positives + negatives) + false(positives + negatives)} \tag{3.1}$$

Precision: Precision (3.2) measures the proportion of files that were correctly classified as faulty over the total number of files classified as either faulty or non-faulty.

$$Precision = \frac{true\ positives}{true\ positives + false\ positives} \tag{3.2}$$

Recall: Recall(3.3) measures the proportion of faulty files which are correctly identified as faulty over the total number of faulty files available.

$$Recall = \frac{true\ positives}{true\ positives + false\ negatives} \tag{3.3}$$

F_1 **Score:** The F1-Score is computed by taking the (weighted) harmonic average of precision and recall as shown in equation (3.4).

$$F_1 Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3.4}$$

Pseudo code: Below in Algorithm 4, is the pseudo code for training and evaluating the model using code and repository metrics. The complete program is in Section A.1.3.

Algorithm 4 Pseudo code to train and evaluate the model using code and repository metrics

```
1: procedure MACHINELEARNINGMODEL    ▷ This program is used to train and evaluate the
   model using code and repository metrics
2:     import libraries    ▷ Eg.pandas,numpy
3:     import dataset    ▷ Eg.code and repository features to train and evaluate the model
4:     Apply Box Cox Transformation    ▷ For skewed features
5:     Draw Pearson Correlation of Features
6:     Split Dataset into Training and Test sets
7:     Re-sample Training Dataset    ▷ To resample imbalanced class by upsampling
   minority class
8:     for each classifier SVC,DecisionTree,AdaBoost,RandomForest,KNN do
9:         Cross validate model with Kfold stratified CV
10:        nsplits=10
11:    end for
12:    initialize votingClassifier    ▷ Using above five models
13:    train votingClassifier
14:    test votingClassifier
15:    print F1Score,Precision,Recall,Accuracy
16:    plot confusion matrix
17: end procedure
```

Chapter 4

Analysis and Obtained Results

In this chapter, we present the obtained results, and we discuss the answers to our research questions RQ1 - RQ5.

4.1 Research Questions RQ1a and RQ1b

RQ1a: Is it possible to train models that can be used to predict fault-proneness of individual projects, by using repository and source code metrics?
*RQ1b: With respect to results from **RQ1a** does the choice of the programming language affect the results?*

Repository metrics: The results pertaining to fault proneness by using information obtained solely from the repository metrics before and after re-sampling the training data is depicted in Tables 4.1a and 4.1b respectively. Considering all the systems as a whole, we obtained an average F1 score of 0.75 with the least being 0.59 for the deeplearning4j system and the highest being 1 for spring-boot and keras systems. Looking at individual systems, the Python based ones yielded F1 scores greater than 0.7, except for the youtube-dl system which yielded an F1 score of 0.68. All Java systems, with the exception of spring-boot, yielded F1 scores less than 0.7.

Code metrics: The results pertaining to fault proneness by using information obtained solely from the code metrics before and after re-sampling the training data is depicted in Tables 4.2a and 4.2b respectively.

For code metrics, considering all the systems as a whole, we obtained an average F1 score of 0.61 with the least being 0.45 for the keras system and the highest being 0.76 for the ipython system. Looking at individual systems, the Python based ones yielded F1 scores greater than 0.7 while the Java based yielded F1 scores less than 0.65.

Aggregated Method: As discussed previously, the two data sets (repository related metrics, and source code metrics) were used independently to train the machine learning system and provide two different models for fault proneness. Each method produces a list of predictions indicated by a flag with *value 1* for files predicted to bear critical errors and a flag *value 0* for files predicted as not bearing errors that can lead to critical failures. The aggregated method intersects the two lists for the files that both have the same flag value (i.e. both have either a 0 or 1 flag value) as depicted in Table 4.3. We then examined the accuracy of the obtained aggre-

Table 4.1: Repository metrics Before re-sampling and After re-sampling.

Project Name	F1 Score	Precision	Recall	Accuracy
springboot	1	1	1	1
deeplearning4j	0.47	0.44	0.5	0.88
elasticsearch	0.59	0.71	0.56	0.95
eclipse-che	0.66	0.73	0.65	0.80
RxJava	0.68	0.68	0.71	0.71
youtube-dl	0.44	0.49	0.41	0.81
ipython	0.77	0.77	0.76	0.84
scikit	0.76	0.77	0.75	0.79
scrapy	0.48	0.47	0.49	0.91
keras	1	1	1	1

(a) Repository metrics (Before re-sampling)

Project Name	F1 Score	Precision	Recall	Accuracy
springboot	1	1	1	1
deeplearning4j	0.59	0.73	0.57	0.87
elasticsearch	0.69	0.75	0.66	0.86
eclipse-che	0.66	0.73	0.64	0.79
RxJava	0.68	0.68	0.71	0.71
youtube-dl	0.68	0.71	0.68	0.72
ipython	0.77	0.77	0.76	0.84
scikit	0.76	0.77	0.75	0.79
scrapy	0.71	0.7	0.73	0.80
keras	1	1	1	1

(b) Repository metrics (After re-sampling)

gated results. These results using the aggregation method of combining results obtained using source code metrics and repository information trained models, before and after re-sampling the training data, are shown in Tables 4.4a and 4.4b respectively.

Considering all the systems combined we obtained an average F1 score of 0.75 with the least being 0.52 deeplearning4j system and the highest being 1 for spring-boot and keras systems. Looking at the systems individually, the Python based ones yielded F1 scores greater than 0.75, with the exception of the scrapy system which yielded a score of 0.61. In contrast the Java based systems yielded F1 scores less than 0.65 with the exception of the spring-boot and RxJava systems.

4.2 Research Question RQ2

RQ2: Given a set of projects $\{P_1, P_2, P_3...., P_n\}$*, written in a language L, is it possible to train the model on all projects* $\{P_1, P_2, P_3...., P_{n-1}\}$ *in order to predict fault proneness for project* P_n*?*

Repository metrics: The results of using a model trained in systems written in one language (i.e. Java or Python) for predicting fault proneness of files in *another* system of the *same* language using only repository metrics are depicted in Table 4.5a. For the results shown, we predicted the fault proneness of the files in the spring-boot system by training the model on the deeplearning4j, eclipse-che, RxJava and elasticsearch systems. The model achieved an accuracy of 0.84 with precision 0.64, recall 0.80 and an F1 score of 0.67.

Similarly, the results on Python systems are shown in Table 4.5b. The model was used to predict the fault proneness of the files in the sci-kit system by training the model on the scrapy,

Table 4.2: Code Metrics Before re-sampling and After re-sampling.

Project Name	F1 Score	Precision	Recall	Accuracy
springboot	0.49	0.48	0.5	0.95
deeplearning4j	0.47	0.44	0.5	0.88
elasticsearch	0.49	0.48	0.5	0.95
eclipse-che	0.43	0.88	0.5	0.75
RxJava	0.63	0.63	0.63	0.70
youtube-dl	0.75	0.99	0.67	0.98
ipython	0.76	0.77	0.75	0.84
scikit	0.67	0.68	0.70	0.68
scrapy	0.48	0.47	0.5	0.93
keras	0.48	0.47	0.5	0.94

(a) Code metrics (Before re-sampling)

Project Name	F1 Score	Precision	Recall	Accuracy
springboot	0.53	0.53	0.53	0.91
deeplearning4j	0.59	0.64	0.58	0.86
elasticsearch	0.56	0.66	0.55	0.83
eclipse-che	0.60	0.62	0.60	0.73
RxJava	0.64	0.64	0.64	0.70
youtube-dl	0.75	0.88	0.74	0.80
ipython	0.76	0.77	0.75	0.84
scikit	0.67	0.68	0.70	0.68
scrapy	0.53	0.53	0.53	0.73
keras	0.45	0.47	0.44	0.82

(b) Code metrics (After re-sampling)

Table 4.3: Example for Aggregated method.

Filename	Prediction(Code)	Prediction(Rep.)	Aggregated?
File1.java	0	0	Yes
File2.java	1	1	Yes
File3.java	1	0	No
File4.java	0	1	No

keras, youtube-dl and ipython systems. These results indicated a prediction accuracy of 0.78 with precision, recall and the F1 score values of 0.78.

Code metrics: The results of training models using only source-code metrics on systems written in the same language (Java or Python), for predicting fault proneness at the file level in *another* system of the *same* language are depicted in Table 4.6a. Here we predicted the fault proneness of the files of spring-boot (Java system), by training the model using only code metrics on the deeplearning4j, eclipse-che, RxJava and elasticsearch systems (Java systems). The model achieved an accuracy of 0.54 with precision 0.54, recall 0.55 and an F1 score of 0.84.

Similarly, the results on Python projects are shown in Table 4.6b. The model was used to predict the fault proneness of the files in the sci-kit system by training the model on the scrapy, keras, youtube-dl and ipython systems. These results indicated a prediction accuracy of 0.69,

Table 4.4: Superimposed (Repository and code metrics) - Before and After re-sampling.

Project Name	F1 Score	Precision	Recall	Accuracy
springboot	1	1	1	1
deeplearning4j	0.47	0.44	0.5	0.88
elasticsearch	0.50	0.49	0.5	0.96
eclipse-che	0.45	0.41	0.5	0.81
RxJava	0.74	0.73	0.75	0.80
youtube-dl	0.49	0.49	0.5	0.98
ipython	0.83	0.83	0.83	0.89
scikit	0.83	0.84	0.83	0.84
scrapy	0.48	0.47	0.5	0.93
keras	1	1	1	1

(a) Superimposed (Repository and code metrics - Before re-sampling)

Project Name	F1 Score	Precision	Recall	Accuracy
springboot	1	1	1	1
deeplearning4j	0.52	0.70	0.52	0.90
elasticsearch	0.56	0.77	0.55	0.89
eclipse-che	0.63	0.80	0.60	0.84
RxJava	0.74	0.73	0.75	0.80
youtube-dl	0.78	0.94	0.73	0.88
ipython	0.83	0.83	0.83	0.89
scikit	0.83	0.84	0.83	0.84
scrapy	0.61	0.61	0.61	0.82
keras	1	1	1	1

(b) Superimposed (Repository and code metrics - After re-sampling)

with precision 0.71, recall 0.70, and an F1 score value of 0.69.

4.3 Research Question RQ3

RQ3: Given a set of projects $\{P_1, P_2, P_3...., P_n\}$, written in languages $\mathcal{L} = \{L_1, L_2, L_3...., L_k\}$ is it possible to train the model on all projects $\{P_1, P_2, P_3...., P_{n-1}\}$ in order to predict bugs for project P_n written in a language $L_i \in \mathcal{L}$?

Repository information: Using Java and Python projects to train the model, the results for predicting file fault proneness on spring-boot (Java based system) by training the model on the deeplearning4j, eclipse-che, RxJava, youtube-dl, ipython, scikit, scrapy, keras and elasticsearch systems are shown in Table 4.7a. The model achieved an F1 score of 0.72 with precision 0.68, recall 0.79 and accuracy of 0.89. Similarly, using Java and Python projects to train the model, the results for predicting file fault proneness on ipython (Python project) by training the model on the deeplearning4j, eclipse-che, RxJava, youtube-dl, spring-boot, scikit, scrapy, keras and elasticsearch systems are shown in Table 4.7b. The model achieved an F1 score of 0.66 with precision 0.66, recall 0.67 and accuracy of 0.69.

Code metrics: Using Java and Python projects to train the model, the results for predicting file fault proneness on the spring-boot system (Java system) by training the model on the deeplearning4j, eclipse-che, RxJava, youtube-dl, ipython, sci-kit, scrapy, keras and elasticsearch systems are shown in Table 4.8a. The model achieved an F1 score of 0.55 with precision 0.54, recall 0.55 and accuracy of 0.85. Similarly, using Java and Python projects to train the

Table 4.5: Predicting for Java project (springboot) and Python project (scikit) using repository metrics.

Training Projects	deeplearning4j, eclipse-che, RxJava, elasticsearch
Test Project	springboot
F1 Score	0.67
Precision	0.64
Recall	0.80
Accuracy	0.84

(a) Predicting for Java project (springboot) using repository metrics

Training Projects	scrapy, keras, youtube-dl, ipython
Test Project	scikit
F1 Score	0.78
Precision	0.78
Recall	0.78
Accuracy	0.78

(b) Predicting for Python project (scikit) using repository metrics

model, the results for predicting file fault proneness on the ipython system (Python system) by training the model on the deeplearning4j, eclipse-che, RxJava, youtube-dl, spring-boot, sci-kit, scrapy, keras and elasticsearch systems are shown in Table 4.8b. The model achieved an F1 score of 0.63 with precision 0.63, recall 0.63 and accuracy of 0.68.

4.4 Research Question 4

Given a set of projects $\mathcal{P}$={$P_1, P_2, P_3 ...., P_n$}, written in the same language L, is it possible to predict fault proneness of a project $P_i \in \mathcal{P}$ by partitioning the projects {$P_1, P_2, P_3 ...., P_n$} into two sets, one set (80%) for training the model and the other set (20%) for predicting fault proneness?

Repository information: Using all the Java projects and repository obtained information alone as features, by randomly selecting the 80% from each project for training purposes, and the remaining 20% for evaluation purposes, the prediction results are depicted in Table 4.9. The model achieved an average F1 score of 0.72 with precision 0.83, recall 0.68 and accuracy of 0.88. Using all the Python projects, by randomly selecting the 80% of each project for training purposes and the remaining 20% for evaluation purposes, the results are depicted in Table 4.9. The model achieved an F1 score of 0.69 with precision 0.67, recall 0.73 and accuracy of 0.81.

Code metrics: Similar to the above, by using all the Java projects and code metrics alone as features, and by randomly selecting the 80% of each project for training purposes and the remaining 20% for evaluation purposes, the prediction results are shown in Table 4.9. The model achieved an F1 score of 0.53 with precision 0.60, recall 0.53 and accuracy of 0.82. Similarly, for the Python projects, by randomly selecting the 80% of each project for training purposes and the remaining 20% for evaluation purposes, the results are shown in Table 4.9. The model achieved an F1 score of 0.79 with precision 0.78, recall 0.81 and accuracy of 0.89.

Table 4.6: Predicting for Java project (springboot) and Python project (scikit) using code metrics.

Training Projects	deeplearning4j, eclipse-che, RxJava, elasticsearch
Test Project	springboot
F1 Score	0.54
Precision	0.54
Recall	0.55
Accuracy	0.84

(a) Predicting for Java project (springboot) using code metrics

Training Projects	scrapy, keras, youtube-dl, ipython
Test Project	scikit
F1 Score	0.69
Precision	0.71
Recall	0.70
Accuracy	0.69

(b) Predicting for Python project (scikit) using code metrics

Table 4.7: Predicting for Java project (springboot) and Python project (ipython) using repository metrics.

Training Projects	deeplearning4j, eclipse-che, RxJava, youtube-dl, ipython scrapy, keras, scikit, elasticsearch
Test Project	springboot
F1 Score	0.72
Precision	0.68
Recall	0.79
Accuracy	0.89

(a) Predicting for Java project (springboot) using repository metrics

Training Projects	deeplearning4j, eclipse-che, RxJava, youtube-dl, springboot, scrapy, keras, scikit, elasticsearch
Test Project	ipython
F1 Score	0.66
Precision	0.66
Recall	0.67
Accuracy	0.69

(b) Predicting for Python project (ipython) using repository metrics

Table 4.8: Predicting for Java project (springboot) and Python project (ipython) using code metrics.

Training Projects	deeplearning4j, eclipse-che, RxJava, youtube-dl, ipython, scrapy, keras, scikit, elasticsearch
Test Project	springboot
F1 Score	0.55
Precision	0.54
Recall	0.55
Accuracy	0.85

(a) Predicting for Java project (springboot) using code metrics

Training Projects	deeplearning4j, eclipse-che, RxJava, youtube-dl, springboot, scrapy, keras, scikit, elasticsearch
Test Project	ipython
F1 Score	0.63
Precision	0.63
Recall	0.63
Accuracy	0.68

(b) Predicting for Python project (ipython) using code metrics

Table 4.9: Predicting bugs by partitioning each project of same language into train and test data.

	Project Name	Language	Train Data(%)	Test Data(%)	F1 Score	Precision	Recall	Accuracy
Repository Metrics	springboot	Java	80	20	0.72	0.83	0.68	0.88
	deeplearning4j		80	20				
	elasticsearch		80	20				
	eclipse-che		80	20				
	RxJava		80	20				
	youtube-dl	Python	80	20	0.69	0.67	0.73	0.81
	ipython		80	20				
	scikit		80	20				
	scrapy		80	20				
	keras		80	20				
Code Metrics	springboot	Java	80	20	0.53	0.60	0.53	0.82
	deeplearning4j		80	20				
	elasticsearch		80	20				
	eclipse-che		80	20				
	RxJava		80	20				
	youtube-dl	Python	80	20	0.79	0.78	0.81	0.89
	ipython		80	20				
	scikit		80	20				
	scrapy		80	20				
	keras		80	20				

4.5 Research Question 5

RQ5: Using advanced code metrics, answer RQ1, RQ2, RQ3 and RQ4. Do advanced code metrics improve the results?

In this experiment, we repeated RQ1, RQ2, RQ3 and RQ4 using advanced source code metrics as shown in the Fig.4.1

Category	Metric name
Cohesion metrics	Lack of Cohesion in Methods 5
Complexity metrics	Halstead Calculated Program Length
	Halstead Difficulty
	Halstead Effort
	Halstead Number of Delivered Bugs
	Halstead Program Length
	Halstead Program Vocabulary
	Halstead Time Required to Program
	Halstead Volume
	Maintainability Index (Microsoft version)
	Maintainability Index (Original version)
	Maintainability Index (SEI version)
	Maintainability Index (SourceMeter version)
	McCabe's Cyclomatic Complexity
	Nesting Level
	Nesting Level Else-If
	Weighted Methods per Class
Coupling metrics	Coupling Between Object classes
	Coupling Between Object classes Inverse
	Number of Incoming Invocations
	Number of Outgoing Invocations
	Response set For Class
Documentation metrics	API Documentation
	Comment Density
	Comment Lines of Code
	Documentation Lines of Code
	Public Documented API
	Public Undocumented API
	Total API Documentation
	Total Comment Density
	Total Comment Lines of Code
	Total Public Documented API
	Total Public Undocumented API
Inheritance metrics	Depth of Inheritance Tree
	Number of Ancestors
	Number of Children
	Number of Descendants
	Number of Parents

Figure 4.1: Advanced Code Metrics

RQ1 In Table 4.10, with the exception of keras, the results for all other projects are not better than both the ones we acquired using basic code or repository metrics.

Table 4.10: Advanced Code Metrics

Project Name	F1 Score	Precision	Recall	Accuracy
springboot	0.52	0.53	0.53	0.67
deeplearning4j	0.53	0.64	0.54	0.82
elasticsearch	0.49	0.64	0.52	0.77
eclipse-che	0.50	0.54	0.51	0.87
RxJava	0.47	0.50	0.50	0.57
youtube-dl	0.55	0.59	0.60	0.55
ipython	0.35	0.40	0.45	0.39
scikit	0.41	0.62	0.52	0.58
scrapy	0.51	0.62	0.54	0.71
keras	0.88	0.97	0.83	0.95

RQ2 In Table 4.11a,4.11b the results for predicting both Python and Java projects were not better than the results acquired by using both basic code or repository metrics.

Table 4.11: Predicting for Java project (springboot) and Python project (scikit) using advanced code metrics.

Training Projects	deeplearning4j, eclipse-che, RxJava, elasticsearch
Test Project	springboot
F1 Score	0.53
Precision	0.54
Recall	0.53
Accuracy	0.85

(a) Predicting for Java project (springboot) using advanced code metrics

Training Projects	scrapy, keras, youtube-dl, ipython
Test Project	scikit
F1 Score	0.59
Precision	0.59
Recall	0.60
Accuracy	0.60

(b) Predicting for Python project (scikit) using advanced code metrics

RQ3 In Table 4.12a,4.12b, combining both Python and Java projects also did not result in better performance compared to using basic code or repository metrics.

RQ4 In Table 4.13, the results again were not better than the ones acquired using basic code or repository metrics.

Table 4.12: Predicting for Java project (springboot) and Python project (ipython) using advanced code metrics.

Training Projects	deeplearning4j, eclipse-che, RxJava, youtube-dl, ipython, scrapy, keras, scikit, elasticsearch
Test Project	springboot
F1 Score	0.55
Precision	0.55
Recall	0.55
Accuracy	0.85

(a) Predicting for Java project (springboot) using advanced code metrics.

Training Projects	deeplearning4j, eclipse-che, RxJava, youtube-dl, springboot, scrapy, keras, scikit, elasticsearch
Test Project	ipython
F1 Score	0.59
Precision	0.66
Recall	0.60
Accuracy	0.65

(b) Predicting for Python project (ipython) using advanced code metrics.

Table 4.13: Predicting bugs by partitioning each project of same language into train and test data.

Project Name	Language	Train Data(%)	Test Data(%)	F1 Score	Precision	Recall	Accuracy
springboot	Java	80	20	0.57	0.74	0.56	0.87
deeplearning4j		80	20				
elasticsearch		80	20				
eclipse-che		80	20				
RxJava		80	20				
youtube-dl	Python	80	20	0.59	0.59	0.59	0.67
ipython		80	20				
scikit		80	20				
scrapy		80	20				
keras		80	20				

Chapter 5

Additional Model Validation in C++ Systems

In this chapter, we present results from an additional set of experiments we have conducted for for the purpose of validating the use of the trained model in systems written in an another language different than Java or Python, which were the languages of the systems the model was trained on.

The objective of this validation is to assess whether the trained model has a more 'general' applicability than on the specific systems and languages used for training it.

In order to conduct this validation, we considered two C++ systems as depicted in Table 5.1. The Kopete system is an industrial strength instant messenger framework that could integrate with a number of systems such as AIM, ICQ, Windows Live Messenger, Yahoo, Jabber, Gadu-Gadu and others.The K3b system is a CD and DVD authoring application by KDE for Unix-like computer operating systems and it provides a graphical user interface to perform most CD/DVD burning tasks.

Table 5.1: Systems used in validating the experiment.

Project Name	Language	Files	Lines of Code	Total Commits (until 2018-09)
kopete	C++	3851	327,281	16,291
k3b	C++	1060	98,933	6,347

5.1 Model Validation Overview

The data collection process was similar to the one used for training the models and as presented in Section 3.3. However for being able to fetch bug reports, we used the official KDE bug tracking system [7]. Figure 3.1 provides an overview of the model validation process. More specifically, as discussed in the previous chapters 3 & 4, the model was generated using 5 Java and 5 Python projects, and with data collected from the corresponding GitHub repositories. The generated model was then used to predict fault proneness in the C++ projects and on data obtained from the KDE repository.

5.2 Results

5.2.1 Using model from RQ3

In this validation, we used the model (using repository metrics only) combining both 5 Java and 5 Python projects. In Table 5.2a, the results are obtained using the model generated from these 5 Java and 5 Python projects and consequently applying the model to the two C++ systems namely, the Kopete and the K3b system. The accuracy, recall and precision for Kopete project is 0.69, 0.78 and 0.61 respectively and for K3b project it is 0.63, 0.62 and 0.62 respectively. For comparison, the results obtained using repository metrics in RQ3 is shown in Table 5.2b.

Table 5.2: Comparing results of data validation on C++ projects with RQ3 repository metrics results.

Project	F1	Accuracy	Recall	Precision
Kopete	0.58	0.69	0.78	0.61
K3b	0.59	0.63	0.62	0.62

(a) Results of the model (combining 5 Java and 5 Python projects using repository metrics) for the two C++ systems.

Project	F1	Accuracy	Recall	Precision
springboot	0.72	0.89	0.79	0.68
ipython	0.66	0.69	0.67	0.66

(b) In RQ3, the results obtained by the model using repository metrics as shown in Table 4.7a,4.7b.

The analysis indicates that the accuracy of the obtained results is much lower when the models are applied to C++ projects. The same trend is also evident with respect to recall and precision. These results may indicate that models trained in an environment of a given system tend to perform better when they are applied to the context of the system they were trained in.

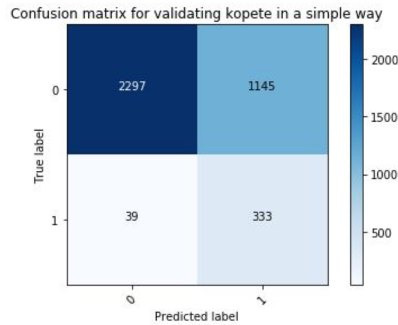
A sample detailed report (for remaining projects see Appendix B.2) for the projects Kopete and K3b is depicted in Figure 5.1a, 5.1b respectively. These data pertain to the results depicted in Table 5.2a.

5.2.2 Using models from RQ2:

In this validation study, we used a model trained in Python projects (using repository metrics only), and a model trained in Java projects (using repository metrics only). Table 5.3a depicts the results for data validation using the model generated from 5 Python projects. The accuracy, recall and precision for the Kopete project is 0.91, 0.75 and 0.75 respectively and for the K3b project it is 0.73, 0.60 and 0.64 respectively. For comparison, the results obtained using repository metrics in RQ2 are shown in Table 5.3b. Here we observe that the model performs quite well compared to the results obtained in RQ2.

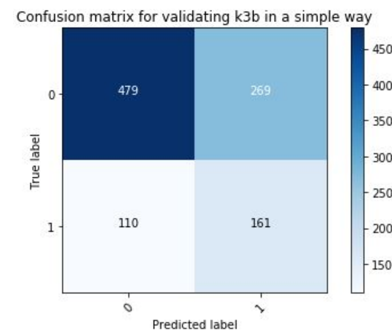
In Table 5.4a, the results are for data validation using the model generated from 5 Java projects. The accuracy, recall and precision for Kopete project is 0.49, 0.71 and 0.58 respectively and for k3b project it is 0.56, 0.61 and 0.59 respectively.For comparison, the results obtained using repository metrics in RQ2 are shown in Table 5.4b. We observe that the performance model in the C++ system is much lower than the performance obtained in RQ2.

```
F1 Score: 0.5775424022152994
Precision: 0.6043046300072293
Recall: 0.7812529287173624
Accuracy: 0.6895647614053487
Confusion matrix, without normalization
[[2297 1145]
 [  39  333]]
```



(a) Kopete detailed report

```
F1 Score: 0.5879366691669236
Precision: 0.5938306945157342
Recall: 0.6172351362551058
Accuracy: 0.6280667320902846
Confusion matrix, without normalization
[[479 269]
 [110 161]]
```



(b) K3b detailed report

Figure 5.1: Kopete and K3b detailed reports

Table 5.3: Comparing results of data validation on C++ projects with RQ2 repository metrics results.

Project Name	F1	Accuracy	Recall	Precision
Kopete	0.75	0.91	0.75	0.75
K3b	0.60	0.73	0.60	0.64

(a) Results of the model (5 Python projects using repository metrics) for the two C++ systems.

Project	F1	Accuracy	Recall	Precision
scikit	0.78	0.78	0.78	0.78

(b) In RQ2, the results obtained by the model (4 Python projects) using repository metrics as shown in Table 4.5b.

Table 5.4: Comparing results of data validation on C++ projects with RQ2 repository metrics results.

Project	F1	Accuracy	Recall	Precision
Kopete	0.44	0.49	0.71	0.58
K3b	0.54	0.56	0.61	0.59

(a) Results of the model (5 Java projects using repository metrics) for the two C++ systems.

Project	F1	Accuracy	Recall	Precision
springboot	0.67	0.84	0.80	0.64

(b) In RQ2, the results obtained by the model (4 Java projects) using repository metrics as shown in Table 4.5a.

The analysis indicated that the accuracy of the obtained results was lower for the Java trained models indicating that these models are not suitable for 'general use' in systems written in another language (in this case C++). For the model trained in Python system we noted a slight increase in accuracy but we cannot conclude that these models are indeed suitable for general use.

Chapter 6

Discussion, Future Work and Conclusion

6.1 Discussion

Based on the experiments we have conducted in this thesis, we are now in a position to provide some answers to our research questions. We classify the result based on F_1 Score *bad*, *weak*, *moderate*, *good* and *very good* using the following conditions.

1) F_1 Score < 0.50 then the prediction capability is *bad* 2) F_1 Score ≥ 0.50 and < 0.60 then the prediction capability is *weak* 3) F_1 Score ≥ 0.60 and < 0.75 then the prediction capability is *moderate* 4) F_1 Score ≥ 0.75 and < 0.85 then the prediction capability is *good* 5) F_1 Score ≥ 0.85 then the prediction capability is *very good*. The overall prediction capability of each different model within the context of the different research questions, is depicted in Table 6.1. Please note, that results for RQ1b are not shown because RQ1b relates to the choice of programming language used.

RQ1a: The analysis of the results depicted in Tables 4.1b, 4.2b and with respect to predicting fault proneness at the file level, indicate that the models trained in individual systems (80% training and 20% testing) using repository information yielded better results, compared to using source code information, for individual system modules. We can conclude that the prediction capability of the model is moderate to good for Python systems and weak to moderate for Java systems.

RQ1b: The analysis of the result depicted in Table 4.1b indicate that repository models trained in Python systems perform better when compared to the ones trained in Java systems. On the other hand, the results of training the models using source code metrics alone in Table 4.2b do not provide a conclusive answer on whether the model performs better for Python or Java systems. Overall, we can say that the aggregated models in Table 4.4b on individual Python projects perform the better as predictors when compared with all other models. Yet, the levels of accuracy, precision, recall and F_1 Score indicate that the overall prediction capability is moderate to good for Python systems and weak to moderate for Java systems.

RQ2: With respect to *Repository metrics*, the analysis of the results depicted in Tables 4.5a and 4.5b, indicate that the predictive capability of the models trained in the same language as

the one that they are applied on, is moderate to good for Python systems and weak for Java systems using code metrics.

With respect to *Code metrics*, the analysis of the results depicted in Tables 4.6a,4.6b indicate that the prediction model is not fit for predicting error proneness on Java systems, having an overall F_1 Score, precision, recall and accuracy values below 0.55. The performance of the model in Python systems was slightly better yielding an F_1 Score, precision, recall and accuracy of 0.69, 0.71, 0.70 and 0.69 respectively. However, we can conclude that code metrics is weak to moderate predictor of fault proneness.

RQ3: With respect to *Repository information*, the analysis of the results depicted in Tables 4.7a,4.7b indicate that models trained in systems written in either language (Java or Python) can serve us moderate predictors for both Java and Python systems.

With respect to *Code metrics*, the results depicted in Tables 4.8a,4.8b, indicate that the prediction capability of the model is weak to moderate for both Python and Java systems. However, the models trained using repository information performed slightly better compared to the models trained using source code metrics.

RQ4: With respect to *Repository information,* the results depicted in Table 4.9 indicate that the model that is trained on 80% of randomly selected parts of the Java systems, and the model trained on the 80% on the randomly selected parts of the Python systems serves as a moderate predictor for both Java and Python systems.

With respect to *Code metrics*, the analysis of the results depicted in Table 4.9 indicate that the model that is trained on 80% of randomly selected parts of the Java systems, and the model trained on the 80% on the randomly selected parts of the Python systems, the prediction capability of the model is good for Python systems, while it is weak for Java systems. Overall, we can say that the code metrics are not a very stable predictor for fault proneness.

RQ5: In addition to the metrics selected, we also conducted a series of experiments using the advanced code metrics shown in Figure 4.1. The results of these experiments are depicted in

Table 4.10 for RQ1; Table 4.11a 4.11b for RQ2; Table 4.12a 4.12b for RQ3 and; in Table 4.13 for RQ4.

These results do not provide any significant new information compared to the results obtained by using the code metrics presented in Section 3.2.2.

Table 6.1: Summary of the discussion

	Language	RQ1a	RQ2	RQ3	RQ4
Repository Metrics	Java	moderate	moderate	moderate	moderate
	Python	moderate to good	good	moderate	moderate
Code Metrics	Java	weak	weak	weak	weak
	Python	weak to moderate	moderate	moderate	good

6.1.1 Thesis Findings

The major findings from this thesis are summarized as follows:

- *Repository metrics were better features to be used than source code metrics.* The possible explanations stemming from this finding are a) programmers often use static analyzers to reduce the software metrics profile of the code they commit, so that they can by-pass automated Quality and Assurance tests - this reduction on the software metrics due to code restructuring will not impact the fault proneness of an already fault prone code; b) for a complex system, often code with high metrics profiles (e.g. high cyclomatic complexity) does not necessarily mean low quality code, as often complex code is used to achieve specific non-functional requirements (e.g. performance) – in this respect metrics alone may not be a good predictor of fault proneness and; c) repository metrics (e.g. number of commits) and other information extracted from DevOps tools (e.g. issues submitted by developers) may provide a better and more "amalgamated" view of the system's health.

- *The predictive models yielded a relatively low F1 score.* The possible explanations stemming from this finding are a) there is noise in the data as we have not factored a weighting scheme whereby more recent data have higher weight that older ones – bugs reported more that two years ago and have been closed may not be relevant for training the system as a whole; b) the tagging of a file as buggy in the training phase was very conservative, as we have considered every issue a developer have flagged as a potential bug, being tagged as a bug for training purposes – often an issue is not necessarily a bug and; c) large systems tend to stabilize over time so in this respect older systems do not provide information-rich data for training after a certain point.

- *The prediction model performed better for Python systems than for the Java systems.* The possible explanations stemming from this finding are a) the Java systems were older so their health profile has been stabilized over time; b) the Python systems have less files which may result to less noise, and; c) the performance difference even though is noticeable may be not statistically significant – experiments with more systems may verify that.

6.1.2 Threats to validity

1. Even though we acknowledged all the perils as mentioned in [43], there is a possibility that results could vary with other projects.

2. The repository metrics and code metrics that we used in our experiment are promising but do not constitute a silver bullet when it comes to finding buggy files.

3. All the project's snapshots in our experiment are taken in the month of February, 2018. So the experiment does not include time related data as it would if we used several snapshots of the same project across multiple releases in the given time period.

4. The machine learning algorithms were used with default parameters without any additional fine-tuning, therefore, fine-tuning the default model could have influenced the results.

6.2 Future Work

The work presented in this thesis can be extended in three major directions. The first direction deals with expanding the training and the analysis by considering more projects and more training features.

The second direction deals with considering more advanced machine learning techniques such as LSTM [27] and by experimenting with fine-tuning the machine learning frameworks.

The third direction deals with assessing technical debt and its relationship to fault proneness.

6.2.1 Additional metrics and projects

1. One possible extension is to experiment with a larger collection of source code metrics to evaluate whether these could outperform the results obtained by using repository data. For example, adding abstract syntax tree extracted from source code of each file as features alongside repository metrics and at the same time expand the analysis to more systems especially the ones that are written in multiple programming and scripting languages. It is also possible to create *embeddings* as metrics which will be extracted from bug reports. The metrics obtained from abstract syntax tree of the source code could be used to create *embeddings* and used along with other metrics.

2. Another possible extension is to consider systems written in other programming and scripting languages. The current system considers systems written in Java, Python and C++ programming languages. One possible extension is to conduct similar study on projects developed using JavaScript, PHP, Ruby, C#, Perl, etc.

6.2.2 Advanced machine learning models

1. Another possible extension in this category is to consider more advanced machine learning approaches and techniques to be used for training purposes. It would also be interesting to see the performance of other machine learning algorithms that we did not use in our experiment.

2. We have used out of the box, default settings for machine learning algorithm. One could fine tune our models so that they can improve the overall performance.

3. T Hall et al. in their study concluded that good predictive performance studies hardly ever optimize their machine learning algorithm settings [35]. Therefore, it would be very interesting to adjust and tweak the algorithms used in this thesis.

6.2.3 Technical Debt

Based on the results, one could further investigate the use of the fault proneness model to identify points where technical debt may be incurred. Some insights on how this can be done are provided in Figures 6.1 and 6.2. In both figures, the x-axis represents time in months, while the y-axis indicates lines of code. Each band represents lines of code added or deleted in a given month. As depicted in Figure 6.1, there are points where a minor or a major refactoring is observed. At this point we can run the prediction model and identify whether files classified as error bearing are now classified as safe and vice versa. This will help assess whether a refactoring operation was successful or not with respect to code debt. Similarly, as depicted in Figure 6.2 a sudden spike may indicate the addition of new code. By applying the prediction model at this point in time we can possibly assess whether the new code incurs defect debt. For example, if 50 files were committed (and there is a spike in the code chart as in Figure 6.2) then we can evaluate using the prediction model, the classification of each of the files as being buggy or not and consequently assess whether these additions incurred a defect debt or not.

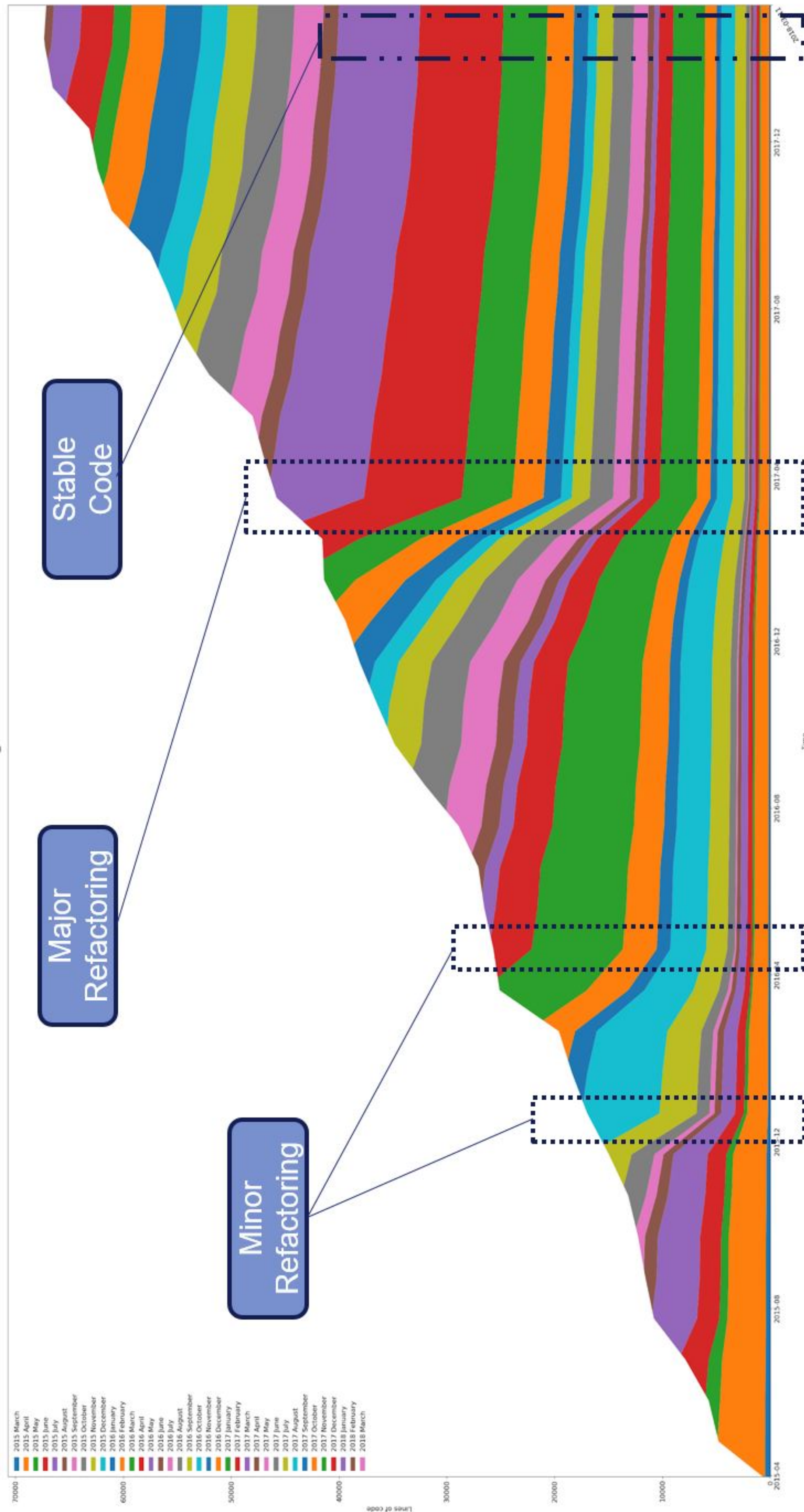


Figure 6.1: Code Debt example [73]

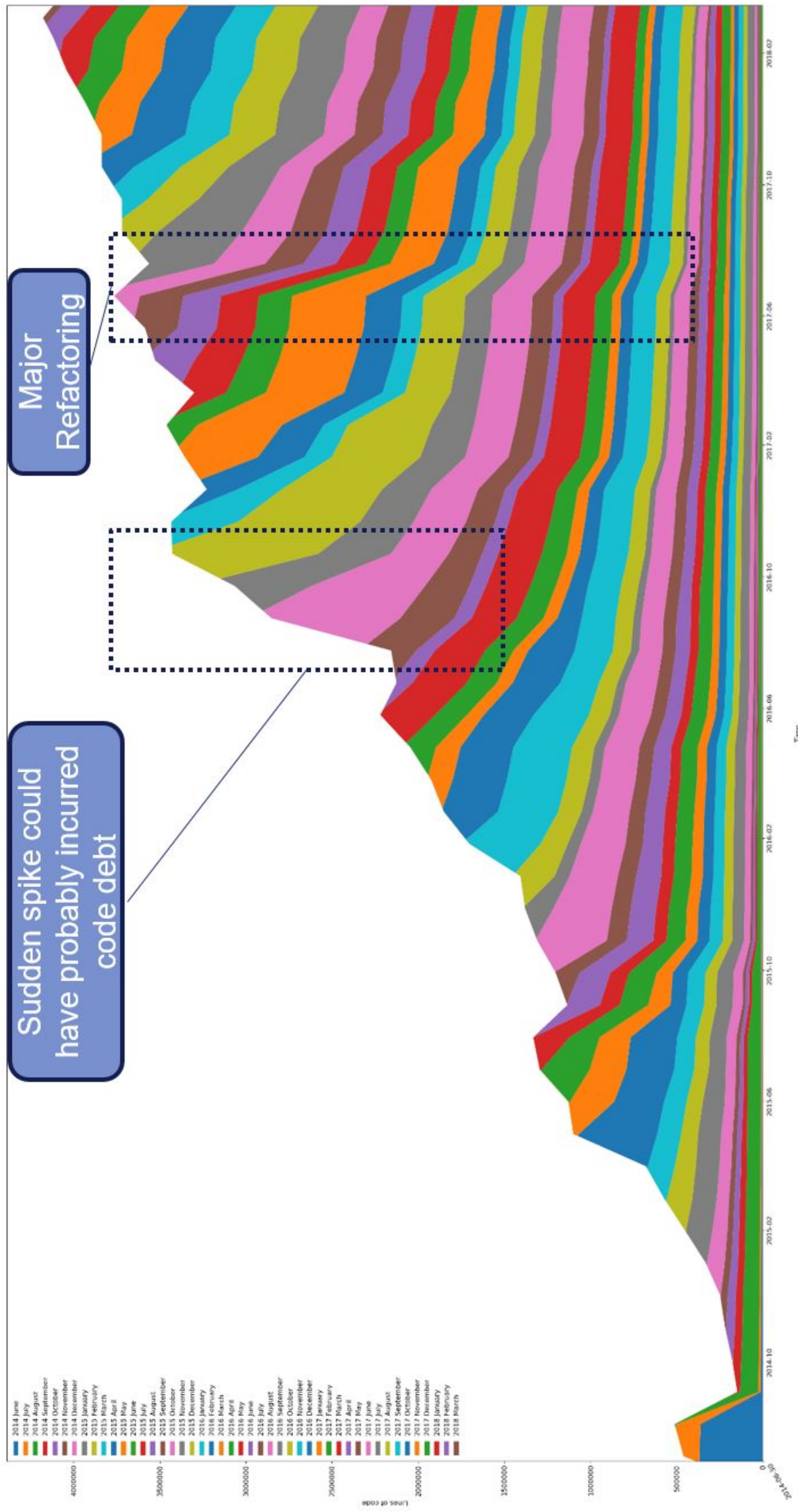


Figure 6.2: Code Debt example [73]

6.3 Conclusion

In this thesis we report results from an evaluation study as to whether information obtained from software repositories and source code metrics can be used as training features in a machine learning framework, so that the trained model can classify a file as one containing a fault (i.e. a bug) that can produce an error and ultimately a failure or not. The evaluation study focused on ten large open source systems written in Java and Python. First, the results indicated that information (Fractal, SOC, Age, and Authors) obtained from repositories and used as training features performed equal to or better than source code metrics (McCC, CLOC, PDA, PUA, LLOC, NOS) in all of the research questions. Second, in all of the scenarios, the prediction capability of the model was weak for Java systems when using source code metrics as a training feature, while it is better when using information extracted from the repository. For Python systems the prediction capability of the model was moderate to good using either source code metrics or repository data as a training feature. Third, the evaluation study indicates that one could use generic training features irrespective of the programming language to train the model, and still perform moderate to good predictions. Finally, by partitioning the data (80% training, 20% test) the model performed very well in both Java and Python systems. It is also noted that the larger the data set for training the model is, the better the prediction capability of the model.

Results using Bugzilla data for K3b and Kopete were similar to the results obtained on GitHub data. The aforementioned repository metrics and source code metrics are not adequate to perform prediction of error proneness at the file level. We need to experiment with additional features and data and possibly consider source code features such as call graphs, as well as uses and sets of variables. Lastly, the study also indicates that this approach is language agnostic because the model was built using Java or Python or a combination of both Java and Python programming languages, but was validated on C++ projects.

Overall, the study indicates that there is potential for further investigation of the use of machine learning for predicting fault prone files in large systems, and facilitating thus testing, maintenance, and evolution activities. Using this study as a springboard, future work includes the following steps. First, one could consider more features and data to be used for training purposes. One could also experiment with a larger collection of source code metrics to evaluate whether these could outperform the results obtained by using repository data, and at the same time expand the analysis to more systems, especially the ones that are written in multiple programming and scripting languages. Second, another possible path is to investigate whether such machine learning techniques can be used to assess or quantify technical debt. More specifically, it would be interesting to investigate whether we can quantify the technical debt incurred by a source code change during maintenance by looking at the altered system features and applying the prediction model to evaluate the error proneness of the components affected.

Finally, one could investigate how machine learning approaches can be combined with static or dynamic code analysis in order to increase accuracy. For example, files which heavily use (e.g. call) fault prone files can be flagged under certain conditions as fault prone themselves.

Bibliography

[1] Github, build software better, together - https://github.com.

[2] Kaggle: Your home for data science, https://www.kaggle.com.

[3] Ndepend: Documentation - https://www.ndepend.com/docs/code-metrics.

[4] Sourcemeter - free-to-use, advanced source code analysis suite, Dec 2016.

[5] Sourcemeter - free-to-use, advanced source code analysis suite, online:https://www.sourcemeter.com/resources/java/, Dec 2016.

[6] Sourcemeter - free-to-use, advanced source code analysis suite, online:https://www.sourcemeter.com/resources/python/, Dec 2016.

[7] Kde bugtracking system, bugs.kde.org, Oct 2018.

[8] Adamtornhill. adamtornhill/code-maat, online:https://github.com/adamtornhill/code-maat, Dec 2017.

[9] Nicolli SR Alves, Leilane F Ribeiro, Vivyane Caires, Thiago S Mendes, and Rodrigo O Spínola. Towards an ontology of terms on technical debt. In *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, pages 1–7. IEEE, 2014.

[10] Tae-Ki An and Moon-Hyun Kim. A new diverse adaboost classifier. In *Artificial Intelligence and Computational Intelligence (AICI), 2010 International Conference on*, volume 1, pages 359–363. IEEE, 2010.

[11] Taiwo Oladipupo Ayodele. Types of machine learning algorithms. In *New advances in machine learning*. InTech, 2010.

[12] Jagdish Bansiya and Carl G Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on software engineering*, 28(1):4–17, 2002.

[13] Boris Beizer. *Software testing techniques*. Dreamtech Press, 2003.

[14] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[15] Barry W Boehm et al. *Software engineering economics*, volume 197. Prentice-hall Englewood Cliffs (NJ), 1981.

[16] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[17] Bora Caglayan, Ayse Bener, and Stefan Koch. Merits of using repository metrics in defect prediction for open source projects. In *Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, pages 31–36. IEEE Computer Society, 2009.

[18] Michelle Cartwright and Martin Shepperd. An empirical investigation of an object-oriented software system. *IEEE Transactions on software engineering*, 26(8):786–796, 2000.

[19] Cagatay Catal and Banu Diri. Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. *Information Sciences*, 179(8):1040–1058, 2009.

[20] Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert systems with applications*, 36(4):7346–7354, 2009.

[21] Ahmad Chaddad, Pascal O Zinn, and Rivka R Colen. Brain tumor identification using gaussian mixture model features and decision trees classifier. In *Information Sciences and Systems (CISS), 2014 48th Annual Conference on*, pages 1–4. IEEE, 2014.

[22] Venkata Udaya B Challagulla, Farokh B Bastani, I-Ling Yen, and Raymond A Paul. Empirical assessment of machine learning based software defect prediction techniques. *International Journal on Artificial Intelligence Tools*, 17(02):389–400, 2008.

[23] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[24] Hassan Chouaib, O Ramos Terrades, Salvatore Tabbone, Florence Cloppet, and Nicole Vincent. Feature selection combining genetic algorithm and adaboost classifiers. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, pages 1–4. IEEE, 2008.

[25] James S Collofello and Scott N Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *Journal of systems and software*, 9(3):191–195, 1989.

[26] Ward Cunningham. The wycash portfolio management system, addendum to the proceedings on object-oriented programming systems, languages, and applications (addendum), 1992.

[27] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. A deep tree-based model for software defect prediction. *arXiv preprint arXiv:1802.00921*, 2018.

[28] Asif Ekbal and Sriparna Saha. Weighted vote-based classifier ensemble for named entity recognition: A genetic algorithm-based approach. *ACM Transactions on Asian Language Information Processing (TALIP)*, 10(2):9, 2011.

[29] Khaled El Emam, Walcelio Melo, and Javam C Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56(1):63–75, 2001.

[30] Norman E Fenton, Martin Neil, and N Square. A critique of software defect prediction models. *SERIES ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING*, 16:72, 2005.

[31] Giles M Foody. Status of land cover classification accuracy assessment. *Remote sensing of environment*, 80(1):185–201, 2002.

[32] Luay Fraiwan, Khaldon Lweesy, Natheer Khasawneh, Heinrich Wenz, and Hartmut Dickhaus. Automated sleep stage identification system based on time–frequency analysis of a single eeg channel and random forest classifier. *Computer methods and programs in biomedicine*, 108(1):10–19, 2012.

[33] David Gray, David Bowes, Neil Davey, Yi Sun, and Bruce Christianson. Using the support vector machine as a classification method for software defect prediction with static code metrics. In *International Conference on Engineering Applications of Neural Networks*, pages 223–234. Springer, 2009.

[34] Dharmendra Lal Gupta and Kavita Saxena. Software bug prediction using object-oriented metrics. *Sādhanā*, 42(5):655–669, 2017.

[35] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[36] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[37] Peng He, Bing Li, Xiao Liu, Jun Chen, and Yutao Ma. An empirical study on software defect prediction with a simplified metric set. *Information and Software Technology*, 59:170–190, 2015.

[38] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 426–437. ACM, 2016.

[39] Xiaolin Huang, Lei Shi, and Johan AK Suykens. Support vector machine classifier with pinball loss. *IEEE transactions on pattern analysis and machine intelligence*, 36(5):984–997, 2014.

[40] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[41] Mohammed J Islam, QM Jonathan Wu, Majid Ahmadi, and Maher A Sid-Ahmed. Investigating the performance of naive-bayes classifiers and k-nearest neighbor classifiers. In *Convergence Information Technology, 2007. International Conference on*, pages 1541–1546. IEEE, 2007.

[42] Nathalie Japkowicz and Mohak Shah. *Evaluating learning algorithms: a classification perspective*. Cambridge University Press, 2011.

[43] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, 21(5):2035–2071, 2016.

[44] Taghi M Khoshgoftaar and Naeem Seliya. Tree-based software quality estimation models for fault prediction. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, pages 203–214. IEEE, 2002.

[45] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90, 2016.

[46] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *Ieee software*, 29(6):18–21, 2012.

[47] Guillaume Lemaître, Fernando Nogueira, and Christos K Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *The Journal of Machine Learning Research*, 18(1):559–563, 2017.

[48] Ruchika Malhotra. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504–518, 2015.

[49] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[50] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, (1):2–13, 2007.

[51] J David Morgenthaler, Misha Gridnev, Raluca Sauciuc, and Sanjay Bhansali. Searching for build debt: Experiences managing technical debt at google. In *Proceedings of the Third International Workshop on Managing Technical Debt*, pages 1–6. IEEE Press, 2012.

[52] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190. ACM, 2008.

[53] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[54] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292. ACM, 2005.

[55] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM, 2006.

[56] Nachiappan Nagappan, Laurie Williams, Mladen Vouk, and Jason Osborne. Using in-process testing metrics to estimate software reliability: A feasibility study. In *Proceedings of IEEE International Symposium on Software Reliability Engineering, FastAbstract, Saint Malo, France*, pages 21–22, 2004.

[57] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. A sloc counting standard. In *Cocomo ii forum*, volume 2007, pages 1–16, 2007.

[58] Ahmet Okutan and Olcay Taner Yıldız. Software defect prediction using bayesian networks. *Empirical Software Engineering*, 19(1):154–181, 2014.

[59] Jason W Osborne. Improving your data transformations: Applying the box-cox transformation. *Practical Assessment, Research & Evaluation*, 15(12):2, 2010.

[60] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Where the bugs are. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 86–96. ACM, 2004.

[61] Matt Parker and Colin Parker. A modified construction for a support vector classifier to accommodate class imbalances. *arXiv preprint arXiv:1702.02555*, 2017.

[62] David Martin Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.

[63] Jane Radatz, Anne Geraci, and Freny Katki. Ieee standard glossary of software engineering terminology. *IEEE Std*, 610121990(121990):3, 1990.

[64] RM Sakia. The box-cox transformation technique: a review. *The statistician*, pages 169–178, 1992.

[65] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 18–27. ACM, 2006.

[66] Carolyn Seaman and Yuepu Guo. Measuring and monitoring technical debt. In *Advances in Computers*, volume 82, pages 25–46. Elsevier, 2011.

[67] Carolyn Seaman and RO Spínola. Managing technical debt. In *Short Course] XVII Brazilian Symposium on Software Quality, Salvador, Brazil*, 2013.

[68] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

[69] Sally Shlaer. The shlaer-mellor method. *Project Technology white paper*, 1996.

[70] Forrest Shull, Vic Basili, Barry Boehm, A Winsor Brown, Patricia Costa, Mikael Lindvall, Daniel Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What we have learned about fighting defects. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, pages 249–258. IEEE, 2002.

[71] Will Snipes, Brian Robinson, Yuepu Guo, and Carolyn Seaman. Defining the decision factors for managing defects: a technical debt perspective. In *Proceedings of the Third International Workshop on Managing Technical Debt*, pages 54–60. IEEE Press, 2012.

[72] Marina Sokolova, Nathalie Japkowicz, and Stan Szpakowicz. Beyond accuracy, f-score and roc: a family of discriminant measures for performance evaluation. In *Australasian joint conference on artificial intelligence*, pages 1015–1021. Springer, 2006.

[73] src d. src-d/hercules, Jun 2018.

[74] Ahmad Taherkhani. Using decision tree classifiers in source code analysis to recognize algorithms: An experiment with sorting algorithms. *The Computer Journal*, 54(11):1845–1860, 2011.

[75] Mie Mie Thet Thwin and Tong-Seng Quah. Application of neural networks for software quality prediction using object-oriented metrics. *Journal of systems and software*, 76(2):147–156, 2005.

[76] Edith Tom, AybüKe Aurum, and Richard Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013.

[77] Adam Tornhill. *Your code as a crime scene: use forensic techniques to arrest defects, bottlenecks, and bad design in your programs*. Pragmatic Bookshelf, 2015.

[78] Burak Turhan and Ayse Bener. Analysis of naive bayes' assumptions on software fault data: An empirical study. *Data & Knowledge Engineering*, 68(2):278–290, 2009.

[79] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, Berlin, Heidelberg, 1995.

[80] Romi Satria Wahono. A systematic literature review of software defect prediction: research trends, datasets, methods and frameworks. *Journal of Software Engineering*, 1(1):1–16, 2015.

[81] Shuo Wang and Xin Yao. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, 62(2):434–443, 2013.

[82] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[83] Bartłomiej Wójcicki and Robert Dabrowski. Applying machine learning to software fault prediction. *e-Informatica Software Engineering Journal*, 12(1), 2018.

[84] Jifeng Xuan, Yan Hu, and He Jiang. Debt-prone bugs: technical debt in software maintenance. *arXiv preprint arXiv:1704.04766*, 2017.

[85] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *QRS*, pages 17–26, 2015.

[86] Nico Zazworka, Clemente Izurieta, Sunny Wong, Yuanfang Cai, Carolyn Seaman, Forrest Shull, et al. Comparing four approaches for technical debt identification. *Software Quality Journal*, 22(3):403–426, 2014.

[87] Nico Zazworka, Rodrigo O Spínola, Antonio Vetro, Forrest Shull, and Carolyn Seaman. A case study on effectively identifying technical debt. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, pages 42–47. ACM, 2013.

[88] Hongyu Zhang. An investigation of the relationships between lines of code and defects. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 274–283. IEEE, 2009.

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

[90] Yong Zhang, Hongrui Zhang, Jing Cai, and Binbin Yang. A weighted voting classifier based on differential evolution. In *Abstract and Applied Analysis*, volume 2014. Hindawi, 2014.

[91] Jun Zheng. Cost-sensitive boosting neural networks for software defect prediction. *Expert Systems with Applications*, 37(6):4537–4543, 2010.

[92] Zhi-Hua Zhou, Jianxin Wu, and Wei Tang. Ensembling neural networks: many could be better than all. *Artificial intelligence*, 137(1-2):239–263, 2002.

[93] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100. ACM, 2009.

Appendix A

Data Extraction Algorithms

A.1 Data Extraction Algorithms:

A.1.1 Algorithm to extract Bug or Issue Data from GitHub:

```
public class IssuesSummary {
 public static void main(String[] args) throws IOException {
  //Setting up github credentials and repository details
  GitHubBuilder builder = new GitHubBuilder();
  GitHub github;
  String oauthToken = "a29e5df4beb31522cd671a92171fbacd6d09119b";
  github = builder.withOAuthToken(oauthToken).build();
  //checking if the connection to repository is successful
  GHRepository ghRepository =


   github.getRepository("spring-projects/spring-boot");
  System.out.println("Connected successfully? -> " +
   github.isCredentialValid() +
   "  Current Repo " + ghRepository.getFullName());

  //Let's try to get all the issues
  List < IssuesWrapper > issuesWrappers =
   new ArrayList < IssuesWrapper > ();
  GHIssueState ghIssueState = GHIssueState.CLOSED;
  PagedIterable < GHIssue > ghIssues =
   ghRepository.listIssues(ghIssueState);
  System.out.println("Size" + ghIssues.asList().size());

  //Iterate over each issue and fetch the commit
  //details of every file that was committed.
  for (GHIssue ghIssue: ghIssues) {
   Collection < GHLabel > ghLabels = ghIssue.getLabels();
   //Most Imp is the Id or IssueNumber
   for (GHLabel ghLabel: ghLabels) {
    if (ghLabel.getName().contains("bug")) {
     IssuesWrapper issuesWrapper = new IssuesWrapper();
     issuesWrapper.issueId = ghIssue.getNumber();
     issuesWrapper.body = ghIssue.getBody();
     issuesWrapper.title = ghIssue.getTitle();
     issuesWrapper.bugLabel = ghLabel.getName();
     GHIssueState state = ghIssue.getState();
     issuesWrapper.state = state.name();

     System.out.println("Added " + issuesWrapper.title +
      " size is " + issuesWrappers.size());
     //add all the details to the wrapper to write
     //it to csv later
```

```
    issuesWrappers.add(issuesWrapper);

    //Trying Again
    GHEventPayload.IssueComment issueeventPayload =
     new GHEventPayload.IssueComment();
    issueeventPayload.setRepository(ghRepository);
    issueeventPayload.setIssue(ghIssue);

    System.out.println(" Action: " +
     issueeventPayload.getAction() + " Comment: " +
     issueeventPayload.getComment());

   }
  }
 }

 System.out.println("Starting CSV Job for Issues");
 CSVWriter csvWriter = null;
 try {
  csvWriter = new CSVWriter(new FileWriter("IssuesSummary.csv"));
  for (IssuesWrapper issuewrapper: issuesWrappers) {
   //Create CSVWriter for writing

   //Date date= new SimpleDateFormat("dd/MM/yyyy").
   //parse(df.format(wrapper.date));
   String[] row = new String[] {
    String.
    valueOf(issuewrapper.issueId), issuewrapper.title,
     issuewrapper.body, issuewrapper.state,
     issuewrapper.bugLabel
   };
   csvWriter.writeNext(row);
  }

 } catch (Exception ee) {
  ee.printStackTrace();
 } finally {
  try {
   System.out.println("Completed!");
   //closing the writer
   csvWriter.close();
  } catch (Exception ee) {
   ee.printStackTrace();
  }
 }
}
}
```

A.1.2 Algorithm to combine Bug or Issue Data with metrics:

```
public class CommitsSummary {

 public static void main(String[] args) {
  DateFormat df = new SimpleDateFormat("dd/MM/yyyy");
  try {
   List < Wrapper > wrappers = new ArrayList < Wrapper > ();
   GitHubBuilder builder = new GitHubBuilder();
   GitHub github;
   String oauthToken = "a29e5df4beb31522cd671a92171fbacd6d09119b";
   github = builder.withOAuthToken(oauthToken).build();

   GHRepository ghRepository =
       github.getRepository("spring-projects/spring-boot");
   System.out.println("Connected successfully? -> " +
    github.isCredentialValid() +
    "  Current Repo " + ghRepository.getFullName());
   PagedIterable < GHCommit > ghCommits = ghRepository.listCommits();
   //System.out.println("Size"+ghCommits.asList().size());

   for (GHCommit ghCommit: ghCommits) {
    ghCommit.getCommitDate();
    ghCommit.getCommitShortInfo().getMessage();
    List < File > files = ghCommit.getFiles();
    if (files != null) {
     for (File file: files) {
      //System.out.println(file.getFileName());
      try {
       Wrapper wr = new Wrapper();
       wr.date = ghCommit.getCommitDate();
       wr.url = ghCommit.getHtmlUrl();
       wr.message = ghCommit.getCommitShortInfo().getMessage();
       wr.filename = file.getFileName();
       wrappers.add(wr);
       System.out.println("Added " + wr.filename +
        " size is " + wrappers.size() + " date is - " + df.format(wr.date));
      } catch (Exception e) {
       System.out.println("Exception occured for id");
      }
     }
    }
   }

   System.out.println("Starting CSV Job for Commits Summary");
   CSVWriter csvWriter = null;
```

```
  try {
   csvWriter = new CSVWriter(new FileWriter("CommitSummary.csv"));
   for (Wrapper wrapper: wrappers) {
    //Create CSVWriter for writing
    //Removed message use above line if you want with message
    String[] row = new String[] {
     df.format(wrapper.date), wrapper.filename, wrapper.url.toString()
    };
    csvWriter.writeNext(row);
   }

  } catch (Exception ee) {
   ee.printStackTrace();
  } finally {
   try {
    System.out.println("Completed!");
    //closing the writer
    csvWriter.close();
   } catch (Exception ee) {
    ee.printStackTrace();
   }
  }

 } catch (IOException e) {
  // TODO Auto-generated catch block
  e.printStackTrace();
 }
}

public static class Wrapper {
 String filename;
 String message;
 URL url;
 Date date;

}

public static class IssuesWrapper {
 String filename;
 Integer issueId;
 String body;
 String title;
 String state;
 String bugLabel;
}

public static class CommitIssueWrapper {
```

```
  String filename;
  String message;
  String url;
  String date;
  String issueId;

 }
}



public class ExtractIssueIdfromCommitsSummary {

 public static void main(String[] args) throws IOException {
  // TODO Auto-generated method stub
  List < CommitIssueWrapper > wrappers =
   new ArrayList < CommitIssueWrapper > ();
  CSVReader reader =
   new CSVReader(new FileReader("CommitSummary.csv"), ',');

  // read line by line
  String[] record = null;

  while ((record = reader.readNext()) != null) {
   String bugNumber = "";
   if (record[2] != null && !record[2].isEmpty()) {
    System.out.println("records - " + record[2]);
    int bugIndex = record[2].indexOf("#");

    if (bugIndex > 0) {
     bugIndex++;

     while (bugIndex < record[2].length() &&
      Character.isDigit(record[2].charAt(bugIndex))) {
      bugNumber = bugNumber + record[2].charAt(bugIndex);
      bugIndex++;
     }
     CommitIssueWrapper ciWrapper = new CommitIssueWrapper();
     ciWrapper.date = record[0];
     ciWrapper.filename = record[1];
     ciWrapper.issueId = bugNumber;
     ciWrapper.message = record[2];
     ciWrapper.url = record[3];
     wrappers.add(ciWrapper);
     System.out.println("bug# - " + bugNumber);
    }
   }
```

```
  }


  System.out.println("Starting CSV Job for Commits Summary");
  CSVWriter csvWriter = null;
  try {
   csvWriter = new CSVWriter(new FileWriter("CommitSummaryPerfect.csv"));
   for (CommitIssueWrapper wrapper: wrappers) {
    //Create CSVWriter for writing
    //Date date= new
        SimpleDateFormat("dd/MM/yyyy").parse(df.format(wrapper.date));
    String[] row = new String[] {
     wrapper.date,
      wrapper.issueId, wrapper.filename,
      wrapper.message, wrapper.url.toString()
    };
    csvWriter.writeNext(row);
   }

  } catch (Exception ee) {
   ee.printStackTrace();
  } finally {
   try {
    System.out.println("Completed!");
    //closing the writer
    csvWriter.close();
   } catch (Exception ee) {
    ee.printStackTrace();
   }
  }
 }
}
```

A.1.3 Algorithm - Machine Learning:

```
import os
print(os.listdir("../input"))

# Load in our libraries
import pandas as pd
import numpy as np
import re
import sklearn
import xgboost as xgb
import seaborn as sns
import matplotlib.pyplot as plt
```

```
%matplotlib inline

#import plotly.offline as py
#py.initnotebookmode(connected=True)
#import plotly.graphobjs as go
#import plotly.tools as tls

import warnings
warnings.filterwarnings('ignore')

# Going to use these 5 base models for the stacking
from sklearn.ensemble import (RandomForestClassifier, AdaBoostClassifier,
   GradientBoostingClassifier, ExtraTreesClassifier)
from sklearn.svm import SVC
from sklearn.crossvalidation import KFold
from scipy import stats
from scipy.stats import norm, skew


import seaborn as sns
%matplotlib inline

from collections import Counter
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier,
GradientBoostingClassifier, ExtraTreesClassifier, VotingClassifier
from sklearn.discriminantanalysis import LinearDiscriminantAnalysis
from sklearn.linearmodel import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neuralnetwork import MLPClassifier
from sklearn.svm import SVC
from sklearn.modelselection import GridSearchCV,
crossvalscore, StratifiedKFold, learningcurve
from sklearn.metrics import accuracyscore, f1score,
precisionscore, recallscore,classificationreport,confusionmatrix,accuracyscore


#read csv files or projects
springboot = pd.readcsv("../input/springboot.csv")

#groupby path and look at data
maindata = maindataungrpd.groupby(['Path']).sum()
maindata.head()


#Lets do Box Transformation
tempForBoxTransformation = pd.concat([tempToReloadProject,
```

```
    tempToReloadOtherProject])
#del tempForBoxTransformation["Path"]
numericfeats =
    tempForBoxTransformation.dtypes[tempForBoxTransformation.dtypes !=
    "object"].index

# Check the skew of all numerical features
skewedfeats = tempForBoxTransformation[numericfeats].apply(lambda x:
    skew(x.dropna())).sortvalues(ascending=False)
print("\nSkew in numerical features: \n")
skewness = pd.DataFrame({'Skew' :skewedfeats})
print(skewness.head(10))

#Transform the skewed features :)
skewness = skewness[abs(skewness) > 0.75]
print("There are {} skewed numerical features to Box Cox
    transform".format(skewness.shape[0]))

from scipy.special import boxcox1p
skewedfeatures = skewness.index
lam = 0.15
for feat in skewedfeatures:
    #alldata[feat] += 1
    tempForBoxTransformation[feat] = boxcox1p(tempForBoxTransformation[feat],
        lam)

tempForBoxTransformation[skewedfeatures] =
    np.log1p(tempForBoxTransformation[skewedfeatures])


#Set it back to repective variable to continue the process
maindata = tempForBoxTransformation[:tempToReloadProject.shape[0]]
del maindata["Path"]
tempToReloadProject = tempForBoxTransformation[:tempToReloadProject.shape[0]]
tempToReloadOtherProject =
    tempForBoxTransformation[tempToReloadProject.shape[0]:]


#Create a column to predict
maindata['predict'] = (maindata['Bug']!=0).astype(int)
maindata.head()

maindata['predict'] = (maindata['Bug']!=0).astype(int)
#del maindata['entity']
maindata.head()
```

```
#Pearson Correlation of Features
colormap = plt.cm.RdBu
plt.figure(figsize=(14,12))
plt.title('Pearson Correlation of Features ', y=1.05, size=15)
sns.heatmap(maindata.astype(float).corr(),linewidths=0.1,vmax=1.0,
          square=True, cmap=colormap, linecolor='white', annot=True)


#split for training and testing
def trainvalidatetestsplit(df, trainpart=.6, validatepart=.2, testpart=.2,
    seed=None):
    np.random.seed(seed)
    totalsize = trainpart + validatepart + testpart
    trainpercent = trainpart / totalsize
    validatepercent = validatepart / totalsize
    testpercent = testpart / totalsize
    perm = np.random.permutation(df.index)
    m = len(df)
    trainend = int(trainpercent * m)
    validateend = int(validatepercent * m) + trainend
    train = perm[:trainend]
    validate = perm[trainend:validateend]
    test = perm[validateend:]
    return train, validate, test

trainsize, validsize, testsize = (80, 0, 20)
train, valid, test = trainvalidatetestsplit(maindata,
                          trainpart=trainsize,
                          validatepart=validsize,
                          testpart=testsize,
                          seed=2017)
ntrain = train.shape[0]
nvalid = valid.shape[0]
ntest = test.shape[0]
print(ntrain,nvalid,ntest);

#Resampling
from sklearn.utils import resample
# Separate majority and minority classes
df_majority = train[train.predict==0]
df_minority = train[train.predict==1]

# Upsample minority class
df_minority_upsampled = resample(df_minority,
                            replace=True,   # sample with replacement
                            n_samples=df_majority.shape[0], # to match
                                majority class
```

```
                                random_state=123) # reproducible results

# Combine majority class with upsampled minority class
df_upsampled = pd.concat([df_majority, df_minority_upsampled])

#assign to main data
train = df_upsampled

#Setting the output for train and test
y_train = train.predict.values

# Display new class counts
df_upsampled.predict.value_counts()
print("Train and Test - Test is original data and not upscaled or included in
    training")
print(train.shape[0])

#remove unnecessary column data
print("Train shapes")
print(train.shape[0])
print(ytrain.shape[0])
print("Test shapes")
print(test.shape[0])
print(ytest.shape[0])

del train["Bug"]
del train["predict"]

del test["Bug"]
del test["predict"]


# Cross validate model with Kfold stratified cross val
kfold = StratifiedKFold(nsplits=10)

# Modeling step Test differents algorithms and checking cross validation
    scores
randomstate = 2
classifiers = []
classifiers.append(SVC(randomstate=randomstate))
classifiers.append(DecisionTreeClassifier(randomstate=randomstate))
classifiers.append(AdaBoostClassifier(DecisionTreeClassifier(randomstate=randomstate),
     randomstate=randomstate,learningrate=0.1))
classifiers.append(RandomForestClassifier(randomstate=randomstate))
classifiers.append(ExtraTreesClassifier(randomstate=randomstate))
classifiers.append(GradientBoostingClassifier(randomstate=randomstate))
classifiers.append(MLPClassifier(randomstate=randomstate))
```

```
classifiers.append(KNeighborsClassifier())
classifiers.append(LogisticRegression(randomstate = randomstate))
classifiers.append(LinearDiscriminantAnalysis())

cvresults = []
for classifier in classifiers :
    cvresults.append(crossvalscore(classifier, train,
        y = ytrain, scoring = "accuracy", cv = kfold, njobs=4))

cvmeans = []
cvstd = []
for cvresult in cvresults:
    cvmeans.append(cvresult.mean())
    cvstd.append(cvresult.std())

cvres = pd.DataFrame({"CrossValMeans":cvmeans,
   "CrossValerrors": cvstd,"Algorithm":
      ["SVC","DecisionTree","AdaBoost",
"RandomForest","ExtraTrees","GradientBoosting",
"MultipleLayerPerceptron","KNeighboors",
"LogisticRegression","LinearDiscriminantAnalysis"]})

g = sns.barplot("CrossValMeans","Algorithm",
     data = cvres, palette="Set3",
     orient = "h",**{'xerr':cvstd})
g.setxlabel("Mean Accuracy")
g = g.settitle("Cross validation scores")


#Using voting classifier
votingC = VotingClassifier(estimators=[('rfc', RFCbest), ('extc', ExtCbest),
('svc', SVMCbest), ('adac',adabest),('gbc',GBCbest)], voting='soft', njobs=4)

votingC = votingC.fit(train, ytrain)

#testing the trained model
testfinal = pd.Series(votingC.predict(test), name="predict")

#results = pd.concat([IDtest,testfinal],axis=1)
#results = testfinal


#tempToReloadProject['predict'] = (tempToReloadProject['Bug']!=0).astype(int)
#t = tempToReloadProject[ntrain:]
print("test shape",test.shape[0])
print("predict shape",ytest.shape[0])
sub = pd.DataFrame()
```

```
#sub['Path'] = t['Path']
#sub['BugActual'] = t['predict']
sub['BugActual'] = ytest#t['predict']
sub['BugPredicted'] = votingC.predict(test)

#storing the results
sub.tocsv("resultsbugs.csv",index=False)

print("F1 Score:",f1score(ytest, sub['BugPredicted']))
print("Precision:",precisionscore(ytest, sub['BugPredicted']))
print("Recall:",recallscore(ytest, sub['BugPredicted']))
print("Accuracy:",accuracyscore(ytest, sub['BugPredicted']))


#confusion matrix
from sklearn.metrics import confusionmatrix
import itertools

def plotconfusionmatrix(cm, classes,
                      normalize=False,
                      title='Confusion matrix',
                      cmap=plt.cm.Blues):
   if normalize:
      cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
      print("Normalized confusion matrix")
   else:
      print('Confusion matrix, without normalization')

   print(cm)

   plt.imshow(cm, interpolation='nearest', cmap=cmap)
   plt.title(title)
   plt.colorbar()
   tickmarks = np.arange(len(classes))
   plt.xticks(tickmarks, classes, rotation=45)
   plt.yticks(tickmarks, classes)

   fmt = '.2f' if normalize else 'd'
   thresh = cm.max() / 2.
   for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
      plt.text(j, i, format(cm[i, j], fmt),
            horizontalalignment="center",
            color="white" if cm[i, j] > thresh else "black")

   plt.tightlayout()
   plt.ylabel('True label')
   plt.xlabel('Predicted label')
```

```
# Compute confusion matrix
cnfmatrix = confusionmatrix(ytest, votingC.predict(test))
np.setprintoptions(precision=2)

# Plot non-normalized confusion matrix
plt.figure()
plotconfusionmatrix(cnfmatrix, classes=[0,1],
                   title='Confusion matrix, without normalization')
```

Appendix B

Machine Learning - Detailed Report

B.1 Cross Validation Scores:

Below is the Cross Validation Scores obtained for projects in RQ1 using repository and code metrics.

RQ1: Cross Validation Score obtained for different projects using Repository Metrics											Average CV Score
SVC	1	0.75	0.8	0.8	0.7	0.7	0.75	0.75	0.8	1	0.805
DecisionTree	1	0.9	0.9	0.8	0.7	0.85	0.8	0.75	0.9	1	0.86
AdaBoost	1	0.9	0.9	0.8	0.7	0.9	0.8	0.7	0.9	1	0.86
RandomForest	1	0.9	0.9	0.8	0.7	0.9	0.8	0.8	0.9	1	0.87
ExtraTrees	1	0.9	0.9	0.8	0.7	0.9	0.8	0.8	0.9	1	0.87
GradientBoosting	1	0.85	0.85	0.8	0.7	0.8	0.75	0.8	0.85	1	0.84
MultipleLayerPerceptron	1	0.65	0.8	0.8	0.7	0.7	0.75	0.75	0.8	1	0.795
KNN	1	0.85	0.9	0.8	0.7	0.75	0.8	0.75	0.85	1	0.84
LogisticRegression	1	0.65	0.8	0.8	0.7	0.7	0.75	0.75	0.8	1	0.795
LDA	1	0.65	0.8	0.8	0.7	0.7	0.8	0.75	0.8	1	0.8

RQ1: Cross Validation Score obtained for different projects using Code Metrics											Average CV Score
SVC	0.6	0.65	0.65	0.65	0.7	0.7	0.75	0.7	0.8	0.75	0.7
DecisionTree	1	0.9	0.9	0.9	0.65	0.8	0.7	0.7	0.9	0.9	0.835
AdaBoost	1	0.9	0.9	0.9	0.65	0.8	0.7	0.7	0.9	0.95	0.84
RandomForest	1	0.9	0.9	0.9	0.65	0.85	0.75	0.65	0.9	0.9	0.84
ExtraTrees	1	0.9	0.9	0.9	0.65	0.85	0.75	0.65	0.9	0.9	0.84
GradientBoosting	0.7	0.85	0.65	0.6	0.65	0.8	0.75	0.7	0.8	0.9	0.74
MultipleLayerPerceptron	0.6	0.65	0.65	0.6	0.7	0.7	0.8	0.7	0.8	0.75	0.695
KNN	0.85	0.85	0.85	0.8	0.65	0.8	0.8	0.65	0.8	0.75	0.78
LogisticRegression	0.6	0.65	0.65	0.6	0.7	0.7	0.75	0.7	0.8	0.8	0.695
LDA	0.6	0.65	0.65	0.6	0.7	0.7	0.8	0.7	0.75	0.8	0.695

Figure B.1: Overview of project selection, research questions, experiment and validation

B.2 Detailed Report:

B.2.1 RQ1:

Repository metrics:

Below is the detailed report for springboot project in Table 4.1b.

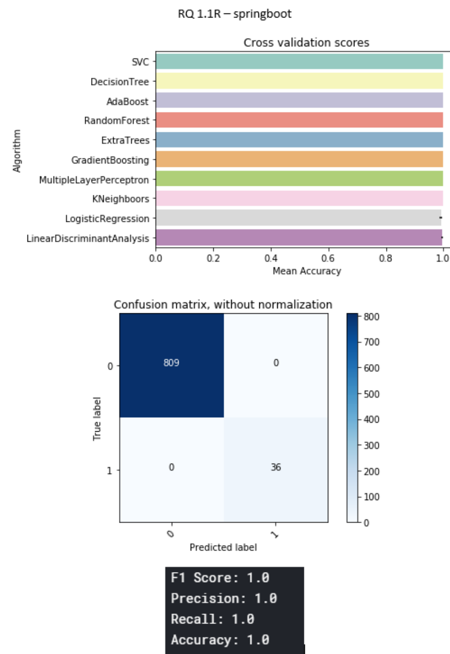


Figure B.2: Springboot project

Below is the detailed report for Deeplearning4j project in Table 4.1b.

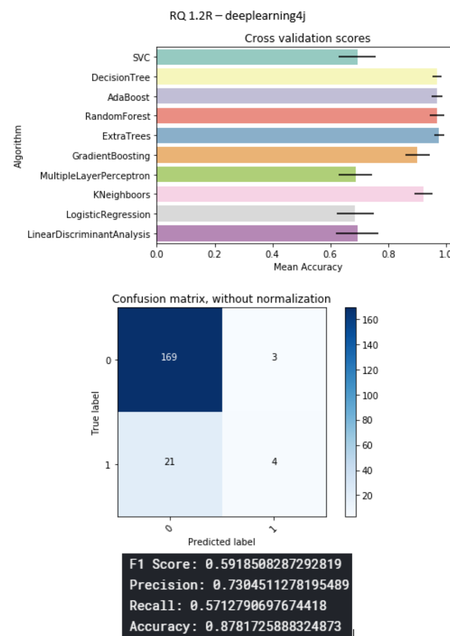


Figure B.3: Deeplearning4j project

Below is the detailed report for Elasticsearch project in Table 4.1b.

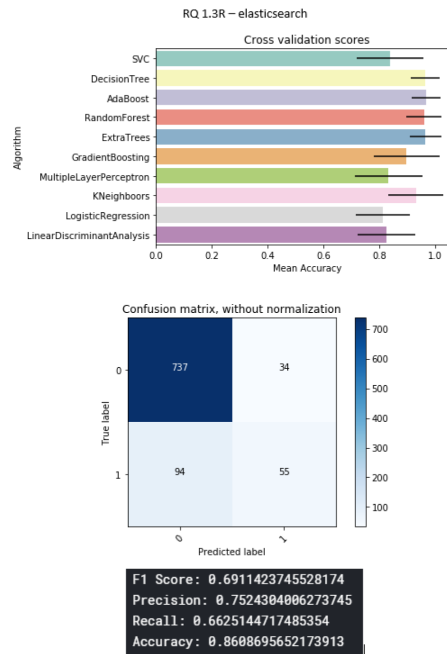


Figure B.4: Elasticsearch project

Below is the detailed report for Eclipse-che project in Table 4.1b.

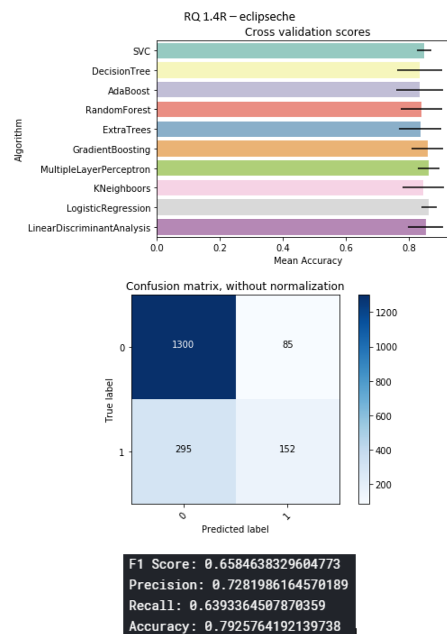


Figure B.5: Eclipse-che project

Below is the detailed report for RxJava project in Table 4.1b.

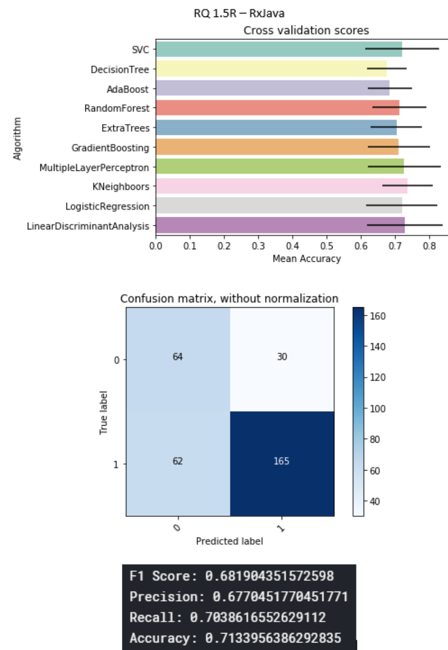


Figure B.6: RxJava project

Below is the detailed report for Youtube-dl project in Table 4.1b.

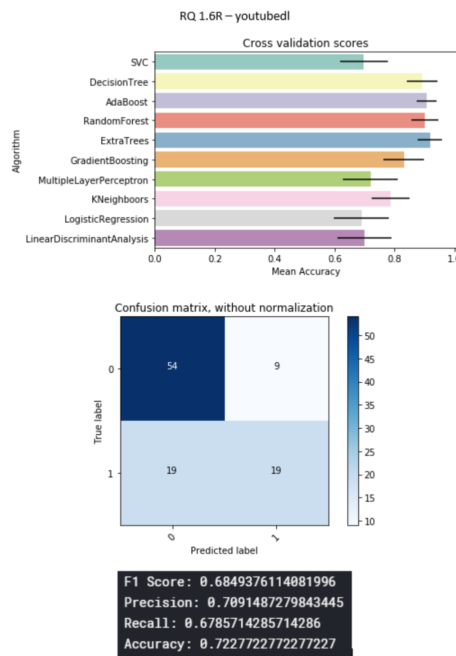


Figure B.7: Youtube-dl project

Below is the detailed report for Ipython project in Table 4.1b.

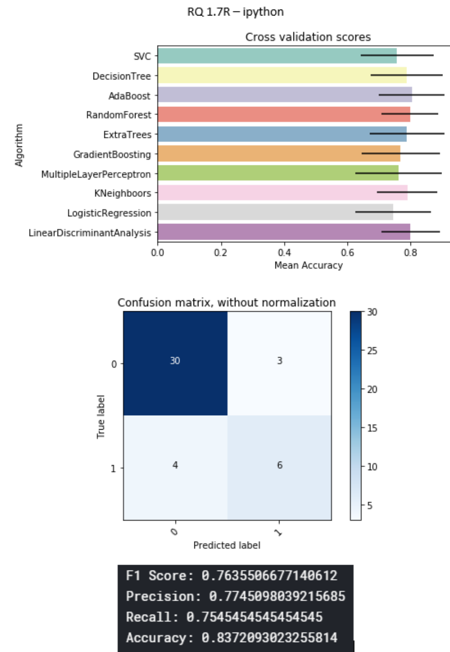


Figure B.8: Ipython project

Below is the detailed report for Scikit project in Table 4.1b.

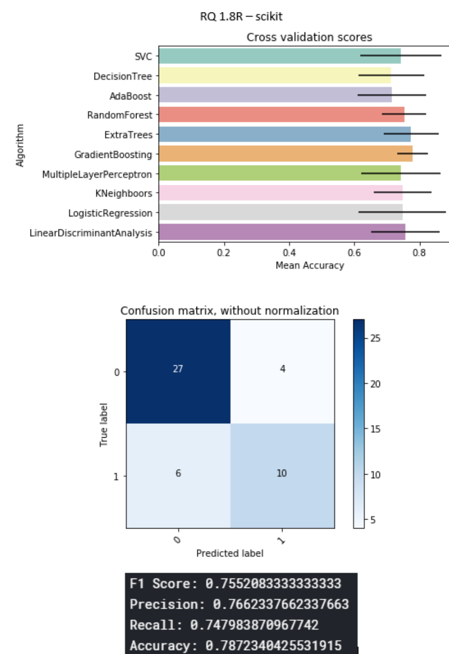


Figure B.9: Scikit project

Below is the detailed report for Scrapy project in Table 4.1b.

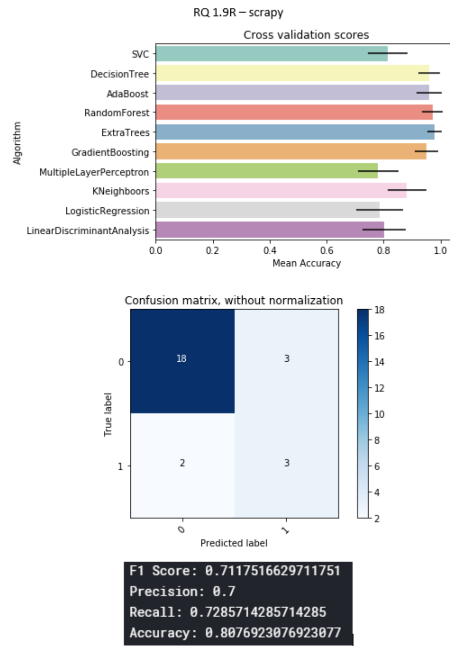


Figure B.10: Scrapy project

Below is the detailed report for Keras project in Table 4.1b.

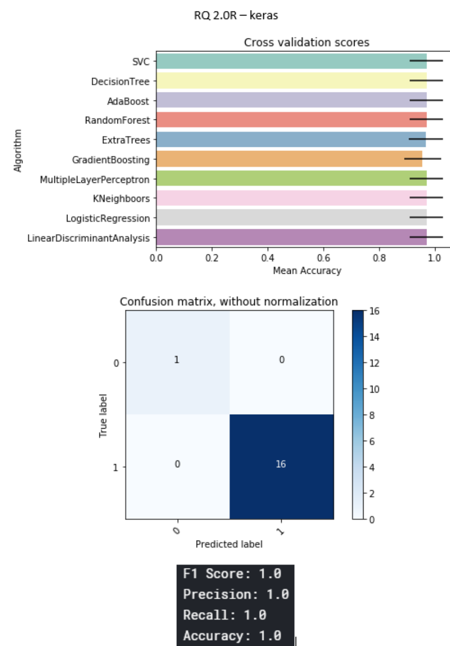


Figure B.11: Keras project

Code metrics:

Below is the detailed report for Springboot project in Table 4.2b.

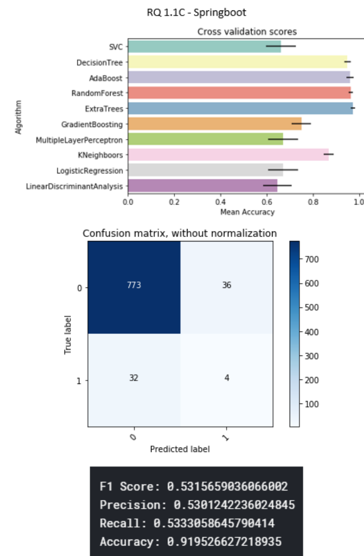


Figure B.12: Springboot project

Below is the detailed report for Deeplearning4j project in Table 4.2b.

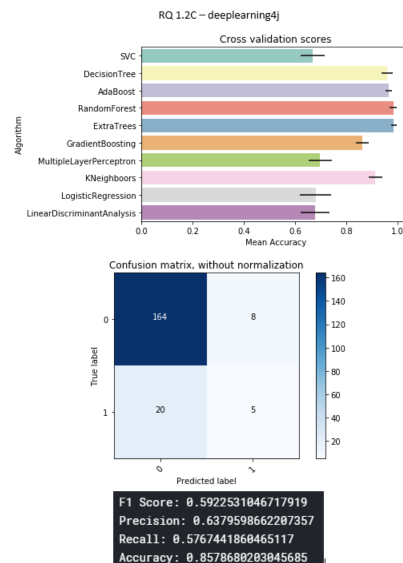


Figure B.13: Deeplearning4j project

Below is the detailed report for Elasticsearch project in Table 4.2b.

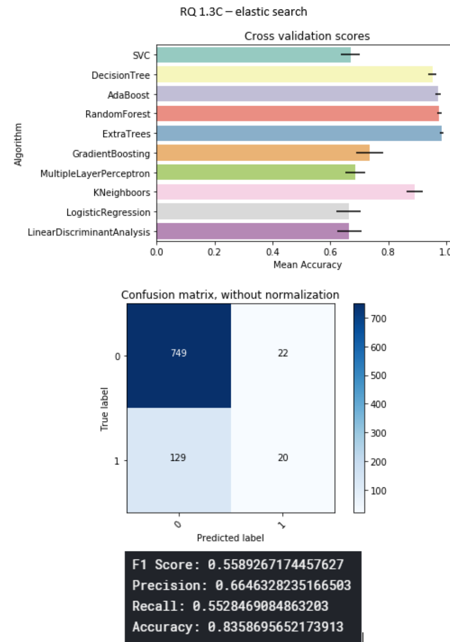


Figure B.14: Elasticsearch project

Below is the detailed report for Eclipse-che project in Table 4.2b.

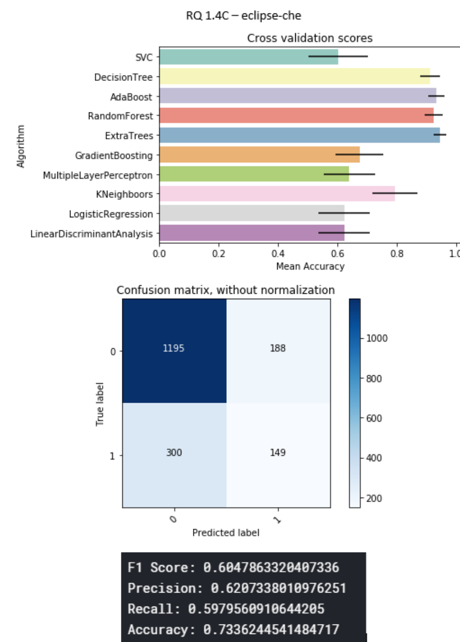


Figure B.15: Eclipse-che project

Below is the detailed report for RxJava project in Table 4.2b.

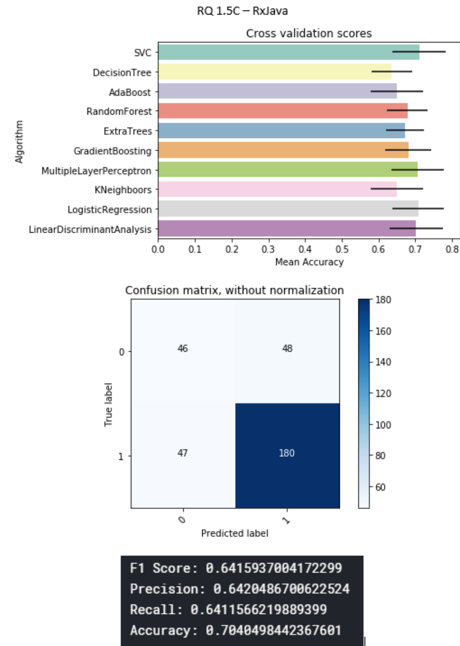


Figure B.16: RxJava project

Below is the detailed report for Youtube-dl project in Table 4.2b.

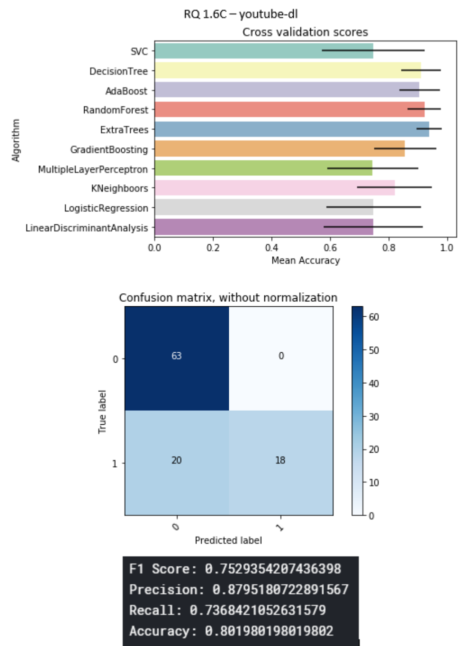


Figure B.17: Youtube-dl project

Below is the detailed report for Ipython project in Table 4.2b.

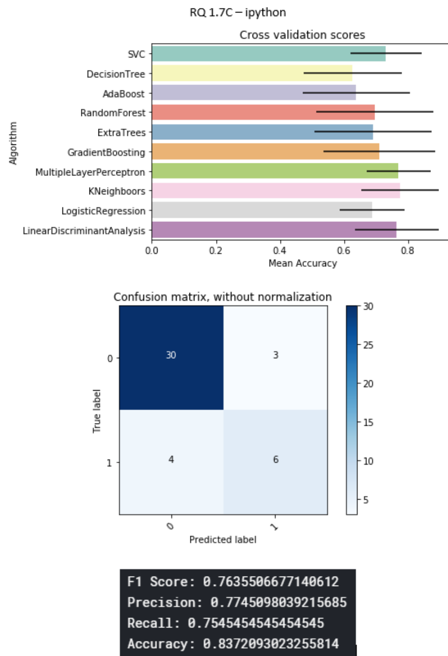


Figure B.18: Ipython project

Below is the detailed report for Scikit project in Table 4.2b.

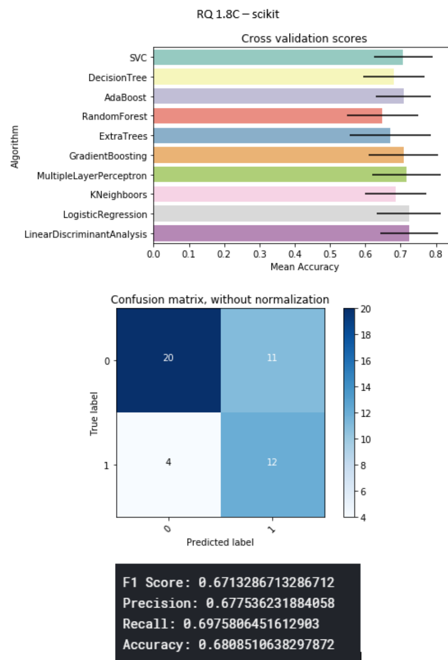


Figure B.19: Scikit project

Below is the detailed report for Scrapy project in Table 4.2b.

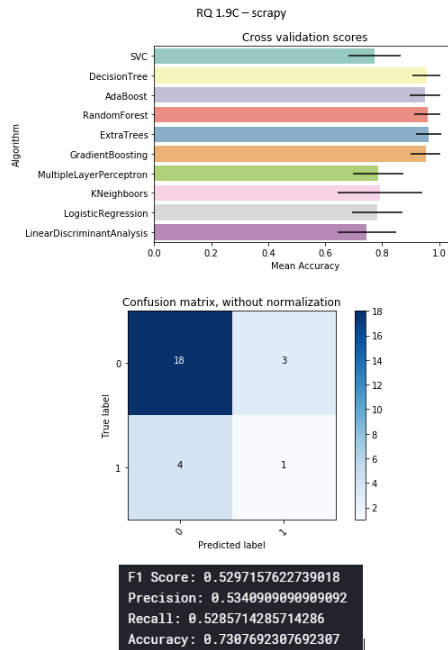


Figure B.20: Scrapy project

Below is the detailed report for Keras project in Table 4.2b.

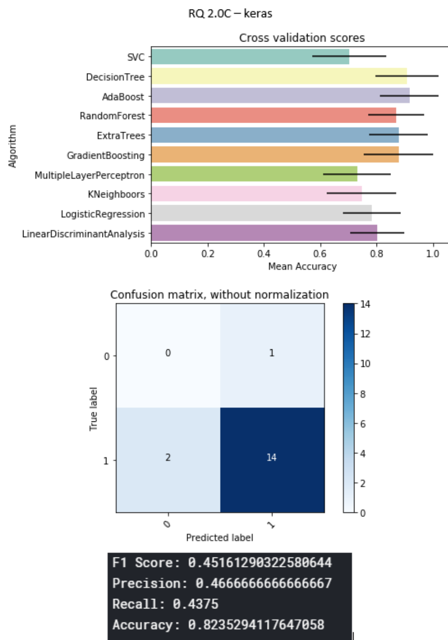


Figure B.21: Keras project

B.2.2 RQ2:

Repository metrics:

Below is the detailed report for java projects using repository metrics in Table 4.5a.

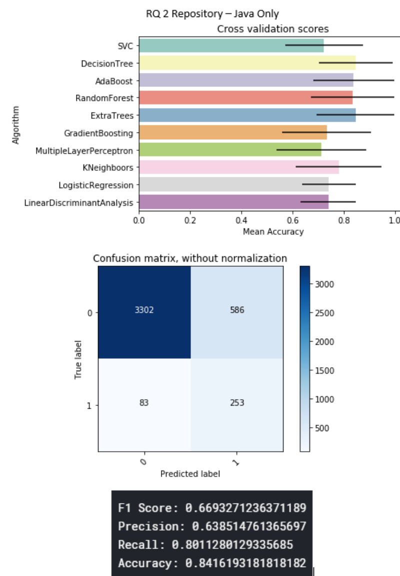


Figure B.22: Repository metrics on java projects only

Below is the detailed report for python projects using repository metrics in Table 4.5b.

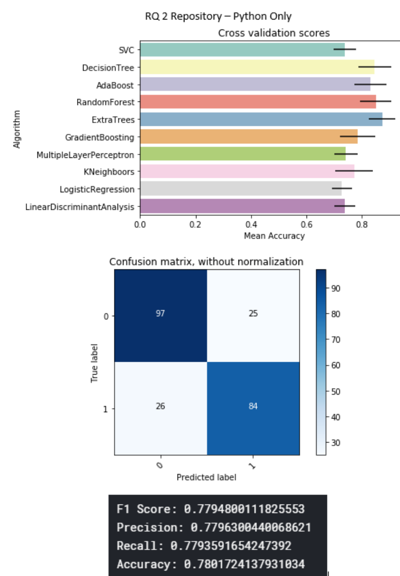


Figure B.23: Repository metrics on python projects only

Code metrics:

Below is the detailed report for java projects using code metrics in Table 4.6a.

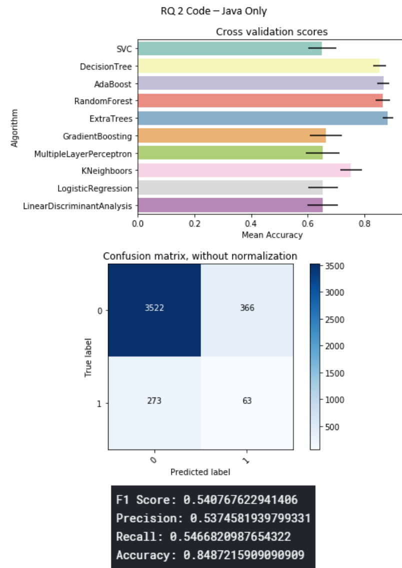


Figure B.24: Code metrics on java projects only

Below is the detailed report for python projects using code metrics in Table 4.6b.

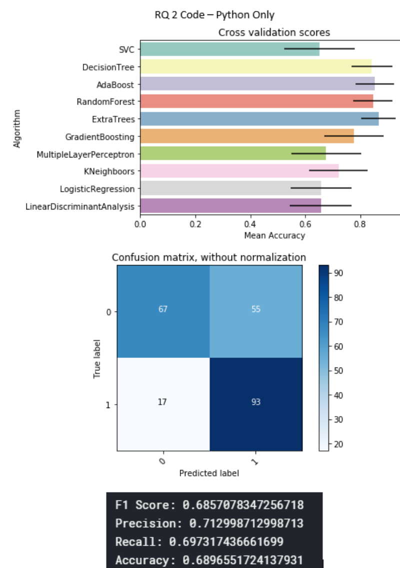


Figure B.25: Code metrics on python projects only

B.2.3 RQ3:

Below is the detailed report for java projects using repository metrics in Table 4.7a.

Repository metrics:

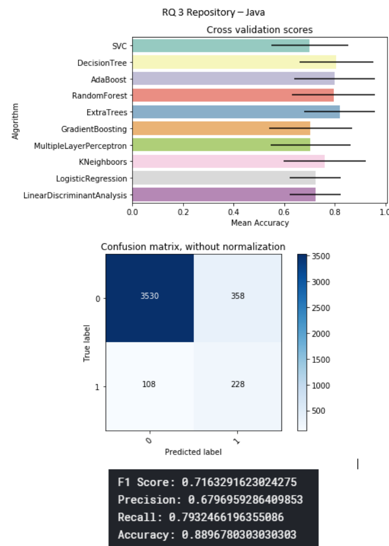


Figure B.26: Repository metrics with java as test project

Below is the detailed report for python projects using repository metrics in Table 4.7b.

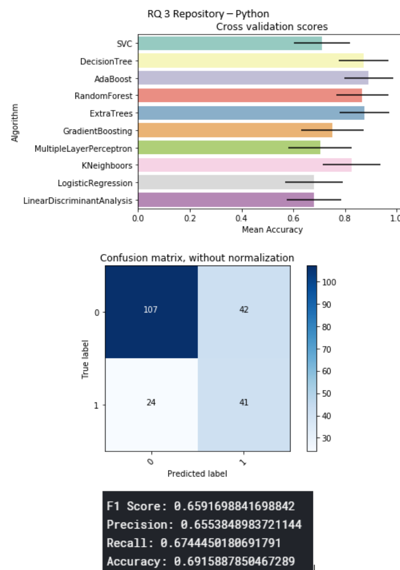


Figure B.27: Repository metrics with python as test project

Code metrics:

Below is the detailed report for java projects using code metrics in Table 4.8a.

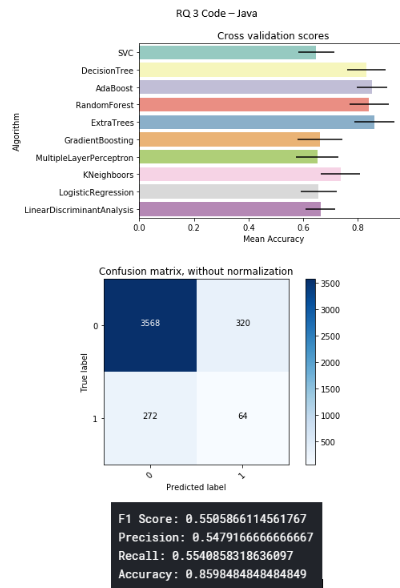


Figure B.28: Code metrics with java as test project

Below is the detailed report for python projects using code metrics in Table 4.8b.

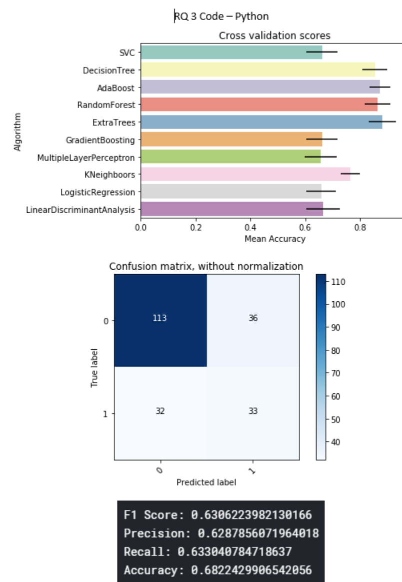


Figure B.29: Code metrics with python as test project

B.2.4 RQ4:

Repository metrics:

Below is the detailed report for java projects using repository metrics in Table 4.9.

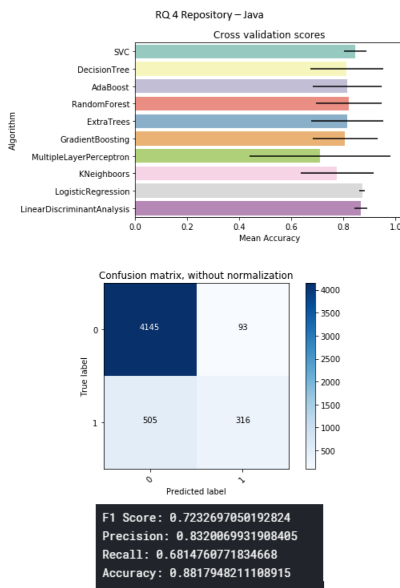


Figure B.30: Repository metrics with java as test project

Below is the detailed report for python projects using repository metrics in Table 4.9.

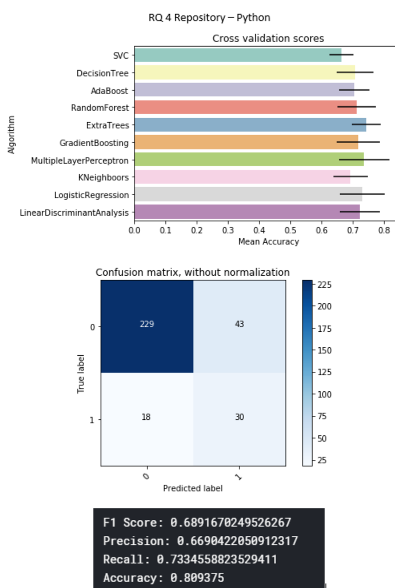


Figure B.31: Repository metrics with python as test project

Code metrics:

Below is the detailed report for java projects using code metrics in Table 4.9.

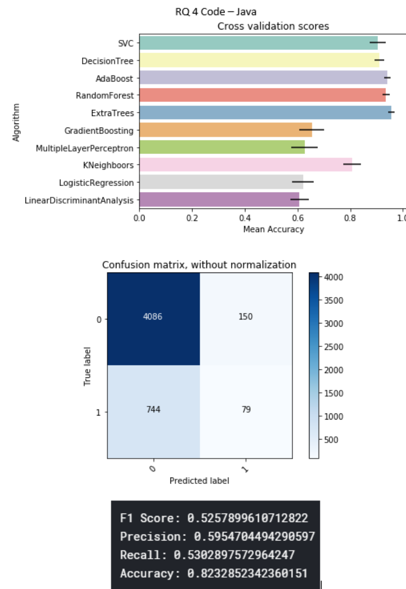


Figure B.32: Code metrics with java as test project

Below is the detailed report for python projects using code metrics in Table 4.9.

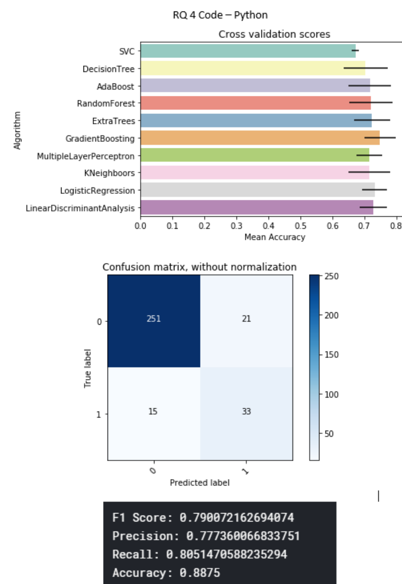


Figure B.33: Code metrics with python as test project

B.2.5 Validation:

RQ2:

Below is the detailed report using Python projects for Kopete and K3b in Figure B.34, B.35 respectively. The results for the same are in Table 5.3a.

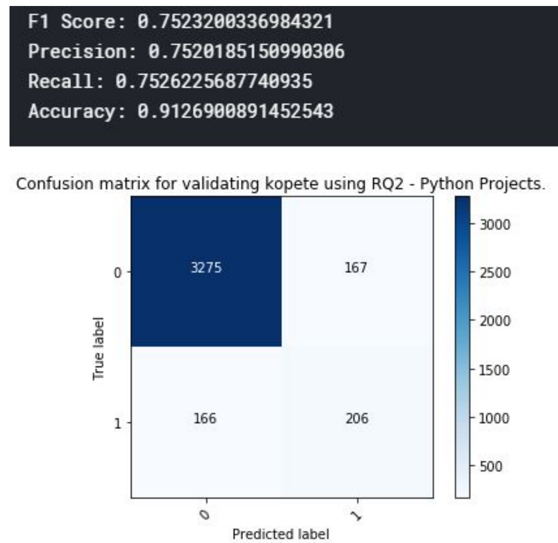


Figure B.34: Kopete detailed report using Python projects

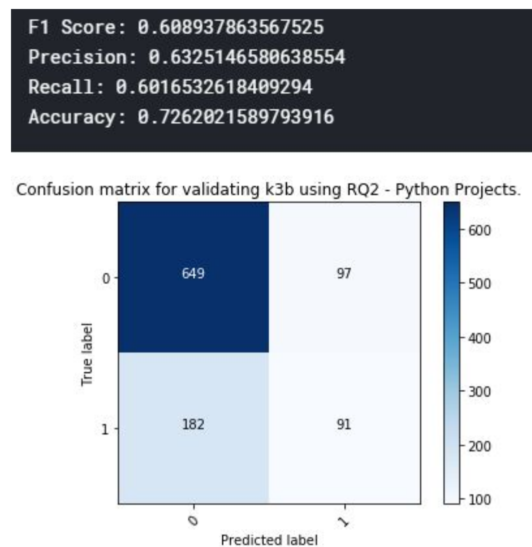


Figure B.35: K3b detailed report using Python projects

Below is the detailed report using Java projects for Kopete and K3b in Figure B.36, B.37 respectively. The results for the same are in Table 5.4a.

```
F1 Score: 0.4370302762475141
Precision: 0.5772457520021375
Recall: 0.7088519115542976
Accuracy: 0.48531725222863137
```

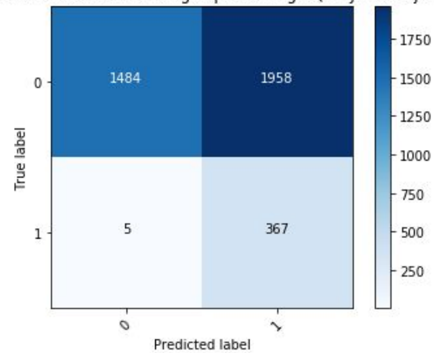


Figure B.36: Kopete detailed report using Java projects

```
F1 Score: 0.5443298354400191
Precision: 0.5856744861169639
Recall: 0.6078130984297204
Accuracy: 0.5583905789990187
```

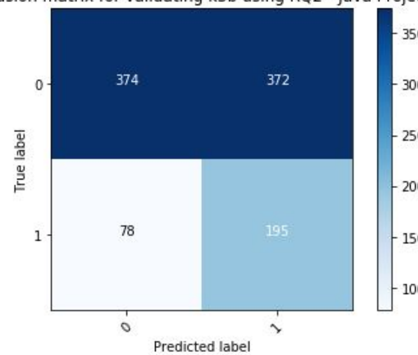


Figure B.37: K3b detailed report using Java projects

Curriculum Vitae

Name:	Sanjay Ghanathey
Post-Secondary Education and Degrees:	Jawaharlal Nehru Technological University Hyderabad, Telangana, India 2007 - 2011 B.Tech.
	University of Western Ontario London, ON 2017-2018 M.Sc
Honours and Awards:	Western Graduate Research Scholarship 2017-2018
Related Work Experience:	Teaching Assistant The University of Western Ontario 2017 - 2018
	Software Developer Intern IBM Software Lab 2018