Fault Driven Supervised Tie Breaking for Test Case Prioritization

by

Vinit Kudva

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Masters of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Regression test suites are an excellent tool to validate the existing functionality of an application during the development process. However, they can be large and time consuming to execute, thus making them inefficient in finding faults. Test Case Prioritization is an area of study that looks to improve the fault detection rates of these test suites by re-ordering execution sequence of the test cases. It attempts to execute the test cases that have the highest probability of detecting faults first.

Most prioritization techniques base their decisions on the coverage information gathered from running the test cases. These coverage-based techniques however have a high probability of encountering ties in coverages between two or more test cases. Most studies employ a random selection to break these ties despite it being considered a lower bound method.

This thesis designs and develops a framework to supervise the tie breaking in coverage based Test Case Prioritization using fault predictor models. Fault predictor models can assist in identifying the modules in the application that are most prone to containing faults. By selecting test cases that cover modules most prone to faults, the fault detection rate of the test cases can be improved.

A fault prediction framework is also introduced in this thesis that supervises the tie breaking for coverage-based techniques. The framework employs an ensemble learner that aggregates results from multiple predictors. To date, no single predictor has been found that can perform consistently on all datasets. Numerous predictors have also required expert knowledge to make them performant. An ensemble learner is a reliable technique to mitigate the problems and bias faced by single predictors and disregard results from poorly performing predictors.

In order to evaluate the supervised tie breaking, empirical studies were conducted on two large scale applications, Cassandra and Tomcat. As part of the evaluation, real faults that existed in the application during development were used instead of hand seeded faults or mutation faults as used by many other studies. The data used for fault prediction were also not groomed or marked by experts, unlike other studies. Results from the studies showed significant improvements in the fault detection rates for both case studies when using the fault driven supervision for tie breaking.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Ladan Tahvildari. If not for her encouragement, I would not have pursued an MASc degree. Her constant guidance and support during my research has been invaluable.

I would like to also thank my committee members, Dr. Lin Tan and Dr. Derek Rayside, for taking time out of their busy schedules to read my thesis. Their feedback has been very helpful and encouraging.

To the past and present members of the STAR group at the university, thank you for your support and guidance in various matters. A special thanks to Sepehr Eghbali for taking the time to clarify details, discussing various parts of the research and providing assistance with the case studies.

I would like to thank my friend, Dr. Andrew Kane, for all the support provided and discussions had during this degree. I have been able to gain new perspectives on the research and clarify ideas based on these discussions.

Finally, I would like to thank my parents, Vasant and Kiran Kudva, and my wife, Anusha, for all their support during this degree.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software systems that exist today are fairly large and can consist of anywhere from a few hundred to a few million of lines of code. Development of these systems is just as complex and requires large effort. Each development cycle requires numerous man hours that frequently get underestimated by the team, resulting in numerous cost overruns [30, 43]. With such level of complexity and tight time constraints, it is inevitable for faults to be introduced into these software systems during the development process. Detecting, locating and rectifying these faults before they become part of a release version of the software system is an expensive and critical path in the development process [13]. If the faults are not corrected before release, the consequences can be quite dire. One recent case is the WannaCry ransomware attack that took advantage of a Microsoft fault and rendered over 300k computers worldwide unusable unless a ransom was paid. The fault that was exploited allowed arbitrary code to be executed on the system thus allowing for the complete system to be encrypted [11].

The presence of faults in software systems, whether severe or minor, is not desirable at any stage of the development cycle. In order to combat the presence of faults in software systems, test suites are developed to validate functionality. The tests within these test suites can be from a number of different categories: unit tests [107], functional tests [37], integration tests [55], etc. The tests not only help detect any faults that might have been introduced, but they also assist with locating the faults and eventually rectifying them. As a result, the test suites along with the quality of tests within the suite is crucial in delivering software that is fault free.

Over time, as new features are developed or existing features are updated, there is a need to verify that all previous functionality is not affected. This verification of previous

functionality is performed using a regression test suite [47]. These regression test suites can either consist of all the coded tests that exist in the application or a subset of them. In some use cases, the test suite can consist of a separate set of tests that have been written by a separate team. No matter the case, as time goes on, the regression test suite for the application tends to grow to an extremely large size in order to be able to verify all the features and functionalities that exist in the application.

Large regression test suites pose the problem of long-run times to verify functionality. This means, developers have to wait an inordinate amount of time to obtain results, thus resulting in large cost overruns. Microsoft estimates the annual cost of regression testing to exceed tens of millions of dollars for all its products [17]. Google also found that a linear increase in commits per day and linear increase in test cases per commit results in a quadratic increase in regression test times [4]. Such cost increases are unsustainable in the long run, resulting in the need to find a way to mitigate the problem of unbounded regression test costs.

One method used to mitigate the problem of waiting an inordinate amount of time for results is to apply a technique known as *Test Case Prioritization* (TCP). The prioritization technique re-orders the execution sequence of the test suite such that test cases with highest probability of finding faults are executed first. This re-ordering should result in faults being detected earlier in the test suite and thus reducing wait times to detect faults. Since the relationship between faults and their revealing test cases is not know before hand, surrogate features of the test cases are used to guide the re-ordering of tests. The surrogate features used are typically statistically or heuristically correlated with exposing faults in code. Most prioritization techniques rely on coverage information of the test cases as a surrogate feature. The coverage information is typically gathered from previous runs of the test suite. The set of techniques that rely on coverage information are aptly known as coverage-based techniques and are an extensively studied area within TCP.

Coverage-based techniques do suffer from a fairly significant problem whereby there is a high probability of coverage ties occurring during the selection of the next test case to prioritize [19]. Breaking these ties to select the next test case is a crucial step in the prioritization algorithm as it can adversely affect the fault detection rate of the test suite. The currently used common method to break these ties is to randomly select a test case from the tied set. This method however is considered to be a lower bound technique that can negatively impact the fault detection rates. Selecting a test case with a lower probability of finding a fault, highly probable in random selection, can cause a significant drop in the fault detection rate of the prioritized test suite. To address this problem, this thesis proposes a fault driven supervised methodology to break the ties. The proposed methodology uses a supervised ensemble learner to generate fault probabilities for all covered classes in the

application and uses this information as another surrogate feature within tie breaking.

Fault prediction is the static analysis of code to understand which modules in the application are at risk of containing faults. Most recent studies have used a class of machine learning algorithms known as classifiers or predictors to predict the faultiness of modules. Studies that have employed classifiers for fault prediction have pointed out that no single classifier can work on all datasets. At the same time, a number of classifiers require expert knowledge to be able to tune them and make them performant. In addition, most of these studies have been performed on pre-groomed or smaller applications that are easier to analyze by a human. To address the problems faced by single classifiers and groomed datasets, this thesis proposes an ensemble learner framework that is able to complete fault prediction without any human intervention. It is the driving force behind the supervised tie breaking that will be used within TCP.

1.1 Problem Statement

TCP at its core is an optimization problem that is an application of the *set cover* and *hitting set* problems. TCP attempts to find the best ordering of test cases by maximizing a set of pre-defined criteria with the aim of improving the fault detection rate. Most approaches use some form of greedy algorithm to achieve the maximization of the pre-defined criteria. A common criteria used for TCP is the coverage achieved by each of the test cases, generally gathered during previous runs of the test cases. TCP algorithms that use coverage as a criteria fall under the category of coverage-based prioritization techniques.

Two techniques come to the forefront of coverage-based prioritization techniques: *Total Technique* (TT) and *Additive Technique* (AT). Given n test cases in a test suite, both techniques iterate n times selecting the next test case to prioritize in each iteration. Within each iteration, TT chooses the next test case that achieves maximum coverage of code components. AT, on the other hand, chooses the test case that has the maximal coverage of code components that have not been covered by previously prioritized test cases [87]. Numerous studies have shown that AT is the superior of the two prioritization techniques, however both techniques tend to suffer heavily from coverage ties [19]. Coverage ties is when two or more test cases achieve the same coverage values and thus a selection cannot be made.

A commonly used technique to break ties in studies is to randomly select a test case from the tied set. This approach to breaking ties is considered to be a lower bound and can negatively impact the fault detection rates. The negative impact is more pronounced in AT

whereby selection of a test case can drastically affect the selection criteria of all remaining iterations. Various studies have proposed augmenting the tie breaking mechanism with surrogate features, thus improving the overall fault detection rate. *Augmenting the tie breaking procedure by supervising it using fault detection classifiers is the first problem this thesis looks to tackle.*

In order to be able to build a tie breaking procedure with fault probability as a criteria, it is necessary to be able to calculate the faultiness of each of the covered classes. The area of study that addresses the fault probability of classes is known as *Fault Prediction or Defect Prediction*. It is a static analysis approach to finding fault trends in the code, using which predictions are made as to where more faults might exist in the code base. Fault prediction methods typically use metrics, a way of quantifying details about the code, as the data source for analysis and predictions. Most recent studies have focused on using classifiers, a subset of machine learning algorithms, to predict the fault-proneness of code components. The numerous classifiers proposed through the various studies have shown very promising results, however they do have a few drawbacks.

Most studies have been performed on datasets that reside in repositories such as PROMISE [68] and MDP [54]. These repositories are fantastic tools to assist with research and comparison studies, however they typically contain pre-groomed datasets. These datasets have been pruned and cleaned along with the faults being clearly marked by various researchers. In some cases, the faults in these repositories have been hand seeded or injected through mutation operators. This is mostly possible due to the fact that most applications within these repositories have a fairly short development history and are only a few hundred modules in size. This approach of grooming data does not however translate very well to large applications as used in TCP research.

Another area of concern with the current approaches to fault prediction is the use of single classifiers. Numerous studies and literature reviews have pointed out that there is no silver bullet when it comes to selecting a single classifier to predict on all applications [103]. Instead, there is a need to understand the application and dataset before hand and then select the classifier that best suites the dataset [62, 99]. There also exists the additional problem that numerous classifiers need expert knowledge to be able to tune them before their results can be useful. The tuning process can also be tedious with the possible need of re-tuning as the application development progresses. Such an approach would not be viable for supervising tie breaking due to the required overhead of maintaining it. Studies have instead shown that using an ensemble learner, an approach to aggregating results from multiple classifiers, is more versatile and consistent in producing results. By aggregating results, the ensemble learner is capable of disregarding any classifier that is under-performing, thus almost always returning a reasonable result.

With the aim of supervising TCP for large applications, the secondary problem this thesis looks to tackle is predicting fault-proneness of components using only their source code history and no human intervention. An approach that is easily adaptable to multiple situations with least amount of effort would be an ideal situation. Being able to predict on large systems and thus assist with TCP tie breaking on these systems would be very beneficial in numerous areas.

So far, two problems have been identified when it comes to TCP and fault prediction. TCP has the problem of ties during test case selection phase of the algorithm and fault prediction in its current state is not a viable solution to supervising TCP tie breaking on large projects. *This leads to the overall problem that the thesis is looking to address: designing and developing a framework for fault driven supervised tie breaking for use in coverage-based Test Case Prioritization.*

1.2 Thesis Contributions

To address the problem of tie breaking in TCP by applying fault probabilities, this thesis makes two main contributions. The contributions are as follows:

Tie Breaking Mechanisms - Two tie breaking mechanisms supervised by fault prediction are proposed.

Fault Prediction Framework - An ensemble learner framework capable of completing fault prediction using the source code repository as its data source and no human intervention at any stage of prediction.

As part of its contributions, this thesis will also look to answer the following research questions:

RQ1 How do tie-breaking schemas that use fault–proneness compare to AT TCP?

RQ2 How viable is it to use a completely automated tool to predict fault-proneness?

RQ3 Is there a correlation between performance of a fault predictor model and the resulting APFD?

Using the two contributions, a complete framework is proposed in this thesis to detect tests, predict faulty classes and finally prioritize tests based on the gathered information.

1.3 Thesis Organization

The rest of this thesis is organized into further 4 chapters. Chapter 2 is a literature review of past works in TCP, tie breaking within TCP and fault prediction. The current state of research will be presented along with the gaps being addressed in this thesis.

Chapter 3 presents the goals of this thesis along with the framework to achieve said goals. The design of the framework being proposed to address the problem of breaking ties in TCP using fault proneness will be presented. There will be a detailed explanation of the various components within the framework along with details about the data flow between components.

Chapter 4 details the implementation of the framework and does an in-depth analysis of the experiments performed on the framework. It will detail the languages, tools and libraries used to realize the framework. The chapter will discuss the applications used as case studies and how the framework performed on each of them. The chapter will also discuss threats to the validity of the results.

Chapter 5 summarizes the works presented in this thesis and presents the conclusions. It will also outline the future works that can be conducted to expand on the findings in this thesis.

Chapter 2

Background and Related Works

In this chapter, past works, state of research and background information will be presented. Section 2.1 will describe TCP and review its relevant history. Section 2.2 will look at the progress in Tie Breaking schemas for TCP. Section 2.3 will discuss details of fault prediction along with its history.

2.1 Test Case Prioritization

Regression test suites are a common technique used to detect faults in software close to the end of the development cycle. It is a test suite that is used to verify that previously existing program functionality has not been affected as new or upgraded features and functionalities have been added to the application [47]. As the application evolves over time, the test suite tends to grow at a fairly quick rate to support all the new features to verify. This causes the test suites to be increasingly expensive to run in terms of time and effort, thus consuming up to 80% of the overall testing budget [60].

Due to the time constraints set on development efforts, cost restrictions and limited resource availability, there is a need to make regression test suites more efficient in detecting faults. There have been three major areas of focus to improve fault detection rates for test suites: Test Suite Minimization, Test Case Selection and TCP [47]. Test Suite Minimization looks to reduce the number of test cases in a test suite by removing any test cases that may be repeating coverage or behaviour performed by other test cases [8]. Test Case Selection attempts to reduce the test suite size by selecting the test cases that are most pertinent to the latest changes in the software [71]. TCP attempts to reorganize the test suite to maximize an objective criteria, typically the rate of fault detection [88].

Focusing on TCP, there has been a considerable amount of research performed on identifying techniques that produce the highest performing test suites after prioritization. There are multiple categories under which TCP techniques can fall. Some of the techniques rely on coverage information, others use historical information and others are based on model or requirement analysis.

Search based test prioritization is a methodology of adding the next test case to the test suite based on a set of predetermined fitness criterion [34,56,59,75]. These fitness criterion are normally a set of requirements about the code being tested which could include details of some sort of code coverage. The techniques used in search-based TCP were heuristic based algorithms such as greedy algorithms and hill climbing algorithms [20, 57]. Over time, the techniques have transitioned towards using meta-heuristic or artificial intelligence based algorithms that could provide better efficiency [61, 81]. Some of the meta-heuristic or artificial intelligence based algorithms used are Ant Colony Optimization and Genetic Algorithm. Quite a few of these techniques have shown promise to perform better than coverage based techniques, however the results have not been conclusive due to the studies being performed on smaller applications and test suites.

Similarity based techniques take into account the similarity between tests, based on a pre-defined criteria, and attempt to diversify the selection of test cases [53]. Miranda et al. [70] propose using hashing based approaches designed to accelerate the prioritization of large-scale test suites. Among black-box similarity based approaches, Ledru et al. [53] propose a similarity based approach that takes into account the strings that describe the test cases, which are the input data or the JUnit test case. Noori and Hemmati [78] also propose a history-based approach in which test cases are prioritized based on how close they are to failing test cases.

Some test prioritization methods do not use the code base of the application to make prioritization decisions. They instead analyze the architecture or requirements of the application and build a set of criterion for prioritization accordingly. For example, Arafeen et al. [7] propose the use of software requirements to cluster and order test cases. Ouriques et al. [80] investigate prioritization techniques that rely on breaking down a system into a behaviour model. The study specifically looks at how the prioritization techniques are affected by the structure of the generated model and the profile of the failing test cases.

A few studies take historical fault data into account too. Laali et al. [51] utilize the locations of detected faults of previously executed test cases to prioritize the rest of the test suite. Yu et al. [105] study a prioritization technique that is based on using fault-based test case generation models. One major assumption in their approach is that the fault-revealing ability of each test cases is known in advance.

Ordering test suites based on the changes made in the application is another area of research that has been garnering attention. The reasoning behind this approach is that covering parts of code that have been modified or are linked to modifications will result in better efficiency of fault detection [22]. [40] used the property that modifications in source code lead to changes in test output as an input criterion to TCP. They managed to split the code into relevant slices that were either directly or indirectly affected by the modifications and then use this information to achieve a promising result.

Most studies though tend to focus on coverage-based TCP techniques. They are amongst the earliest used techniques for prioritizing test suites and can be considered a simplified version of search based techniques. The foundation of this methodology is that to be able to discover a fault there needs to be a test case that can execute the faulty code. Coverage-based techniques use various granularities of code coverage (branch, class, statement, method) as the criterion for prioritization [16, 88]. The coverage is gathered by instrumenting the code to determine which statements, branches or units of code of the application are executed by each test. The most common coverage criterion used in studies are class, method or statement level coverage. Researchers have also explored other coverage criteria such as definition-use association coverage, modified decision coverage [42], specification coverage [49], static coverage [66] and techniques that account for test case execution time [100].

When looking back at all the research that has been done over the years on coverage-based techniques, it will be noticed that most of the techniques fall into two main categories: Total Technique and Additive Technique. In Total Technique, the test cases are ordered in descending order by the total coverage each test case achieves. In Additive Technique, the test cases are ordered by total coverage achieved without covering the same units of code as the previous test cases in the test suite, in other words, find the next test that gives maximum additional coverage. The units of code in this case could be statement, branch, function or any other code structure. Several studies have shown that the techniques from the category of Additive Technique, including the original technique by Rothermel et al. [87], continue to be the superior techniques available to researchers [32, 33, 57, 106].

There also exists a set of techniques that are considered to be multi-criteria. Depending on the application of the multi-criteria technique, the idea is to start with a single criteria for prioritization and then adding on other criterion, as needed, to either break ties or switch between criteria dynamically to better suite the needs of the test suite [71]. The aim of multi-criteria is to achieve better consistency and results across multiple applications using the same criterion, which has been proven in a few studies when compared to single criteria TCP techniques [47, 91]. One such study was a quality-aware TCP, whereby coverage and fault information of the application were used in conjunction to generate prioritized test

suites that performed better than coverage-based techniques [102]. The study used a static analysis technique, FindBugs [36], and an unsupervised technique, CLAMI [76], to detect fault proneness of code and used those fault proneness values to add weights to the classes for final prioritization.

One of the concerns that arises with TCP studies is the data used for verification. The verification data used in most studies over the years consist of hand-seeded faults and/or mutation injected faults. Hand seeded faults, as the name suggests, are faults that have been purposely injected into an application by a developer or researcher to imitate faults that could get introduced in the real world [18]. Mutation injected faults are modifications done to the source code to introduce a fault based on a set of automated rules that are meant to change the application's behaviour [18]. Hunsook Do et al. [18] point out that further studies need to be completed to compare hand-seeded faults and mutation injected faults to better determine the correlation between the two techniques. This also begs the question of how do these two data sources compare to real faults in TCP validation.

Studies have been conducted to gauge the relationship of mutation faults to real faults. Rene Just et al. concluded that there is a significant correlation between real faults and mutation faults in their study [44]. This further brought up the question of whether test suites had an easier time detecting real faults or mutation faults. A study conducted by David Paterson et al. addressed this question and found that prioritized test suites have a harder time to detect real faults compared to mutant faults [82]. The main reason for the result was that mutant faults typically span only one line of code, however real faults have a tendency to span multiple lines of code, thus making them harder to detect. Based on the finding, David Paterson et al. recommended updating the mutation injection rules to contain rules that generate faults by modifying multiple lines of code in order to better facilitate TCP validation. These results make it more apparent that using real fault information from applications is the best way to validate results whenever possible.

In order to be able to evaluate and compare prioritization techniques, there needs to be some method to measure the effectiveness of the technique under study. A common metric used to measure the effectiveness of a technique is Average Percentage Faults Detected (APFD), introduced by Rothermel et al. [87]. It measures the weighted average of the percentage of the faults detected over the lifetime of the test suite and is a value between 0% and 100%. The closer the value to 100%, the better the rate of fault prediction by the test suite. Formally, let T be the set of w test cases and TF_i be the index of the first test case that detects the i-th fault. Also, assume the number of faults that can be detected by the test suite is q, then the value of APFD of an ordering is given by the following equation:

$$\text{APFD} = 1 - \frac{TF_1 + \ldots + TF_q}{wq} + \frac{1}{2w}. \tag{2.1}$$

Higher APFD values signal that the average value of TF_is is lower and thus indicate higher rates of fault detection. A higher rate of fault detection translates to quicker code validation and thus cost savings in the long run.

2.2 Tie Breaking in TCP

Another concern that exists with regards to TCP is that of ties occurring during the selection of the next test case to add to the test suite. Ties occurring are more prevalent in coverage based techniques, however they are possible in almost all techniques. The number of steps that result in a tie in coverage-based techniques are much higher than expected. One study discovered that over 90% of the steps could result in a tie depending on the coverage criteria used [19].

Breaking these ties is a definite concern as one wrong tie breaking result could adversely affect the final fault detection rate of the prioritized test suite. A commonly used technique in studies to break ties is to randomly select a test case from the tied set, but it is also known to be a lower bound technique [41, 79, 90]. Leaving the selection of a test case to chance is never a good idea especially when test suite execution and fault identification is a very expensive process.

Limited studies have been conducted so far to address the problem of ties within TCP. Most of the studies conducted continue to use some form of coverage criteria to break the ties. Eghbali et al. [19] propose a tie breaking schema for AT whereby the complete coverage of the test case is taken into consideration and not just the code that has not been covered yet. Mei et al. [67] propose a technique of using multiple levels of criteria based on detail and add more coverage details when ties occur until the tie is broken.

A few other tie breaking techniques do exist that do not rely on coverage information. Gonzalez-Hernandez et al. [25] propose a TCP technique that relies on killing mutants in the code. The TCP technique ranks all mutants based on how many test cases can detect them and selects the next test case based on maximum mutants that are killed at the lowest rank possible. When a tie occurs, the rank is increased for all in the tied set and the procedure repeats itself. However, it can be noticed that again the main criteria used by the procedure is manipulated to help break ties, number of mutants killed in this case. In addition, this procedure uses mutation faults as its main criteria whereby we know from

other studies that mutation faults do not completely represent real faults in code. He and Bai [35] also proposed a TCP algorithm for GUI testing where they use GUI state coverage as a criterion. When they face a tie though, they just modify the state coverage criterion to break the ties.

As can be noted from all these studies, the tie breaking scenarios typically modify the primary criterion for prioritization. There does not appear to be much research into using a secondary set of criterion that is not related to the primary set. Multi-criteria TCP techniques have shown mixed results with regards to the fault detection rates of the prioritized test suites. There is certainly a need to investigate whether applying multi-criteria for tie breaking would be beneficial

2.3 Fault Prediction

Being able to detect and predict faults using static analysis within software systems has been an area of study for a large number of years. Static analysis is a technique to understand the code without the need for running it by either generating some form of metrics or breaking it down into descriptive structures like trees. Studies in the area of fault prediction started as far back as 1993 with Moller et al. studying distribution of software faults [73]. The study found that fault distribution is not uniform across the code. Majority of the faults within software systems occur in a small percentage of the code base.

Fault prediction algorithms have a very long history. Some early studies used statistical models such as binary logistic regression, univariate regression and multivariate regression to predict faults [9,92]. This set of methodologies are considered to be a black-box and suffer from the problem of needing a lot of data to have any significant result. In addition, these models are not transferrable between organizations or applications thus making them very expensive to use [9]. In other words, if one organization happened to create and validate a model, another organization or even another application would not be able to use the model for their own fault prediction analysis.

Another set of techniques that is widely used for fault prediction is rules based techniques [36]. The rules used to detect faults are typically patterns a developer might use that are known to result in faulty code and can be detected by performing source code parsing analysis. A similar concept is that of code smells, which are design patterns that are prone to faults but might not be faulty to start with [45]. Such techniques though are dependent on the knowledge of the language being used and the bad code patterns

that exist within the language. They can however be used to augment other techniques to better predict faults in code [45].

In the last decade, research has turned to using artificial intelligence and machine learning algorithms to help predict faults in code. A variety of machine learning techniques, supervised and unsupervised, have been successfully applied to help predict faults. A supervised algorithm in fault prediction applications typically use fault data from previous releases or similar applications to form a model, whereas an unsupervised algorithm does not need any defect data and tries to cluster modules with similar attributes [6]. The unsupervised algorithm would need an expert to label the various clusters as faulty or non-faulty though, which either needs some human intervention or needs some sort of statistical analysis to be applied.

In the unsupervised learning category of machine learning, one popular tool to predict software faults is CLAMI [76]. It works by clustering the classes based on their metrics and then labels the clusters using statistical analysis. The statistical analysis consists of splitting the metrics into high and low values based on a median and then labelling the higher values as faulty. The reasoning for this was that higher code complexity results in higher fault-proneness [69]. Such an approach is very useful, however, large complex applications will typically have higher complexity values in general thus possibly skewing the results. In addition, [102] found that CLAMI did have a higher false positive rate, which can be of concern in large applications. Similarly, [23] found that unsupervised algorithms are affected by the datasets they are used on and thus are not universally acceptable to be as good as supervised learning algorithms.

Supervised learning techniques have been the more popular choice in fault prediction research. Some of the most common supervised machine learning algorithms that have been used so far are Logistic Regression, Naïve Bayes and Neural Networks [99]. The first two algorithms have been used extensively and have been deemed to be reliable in detecting faults [69]. There are other techniques though, like Support Vector Machines that do not perform well until they are tuned correctly for the given dataset [28, 94]. Once all the algorithms have been tuned though, it has been noted that there is not much of a performance difference between the various algorithms [54, 64]. However, [99] has noted that choosing the right algorithm for the right dataset is still key to getting good results for fault predictions.

Supervised learning algorithms have their own set of problems though when it comes to fault prediction. The first problem is the *no free lunch theorem*, whereby there is no predictor that can universally perform on all datasets [103]. In other words, one needs some form of expert knowledge to be able to match a given dataset to a predictor in order

to obtain beneficial results. The other problem with datasets in fault prediction is that they are typically imbalanced [21, 85]. An imbalanced dataset is where one category in the dataset has significantly more data points within it compared to the other categories. The imbalance in fault prediction datasets is towards non-faulty data as there are a larger number of non-faulty data points compared to the faulty data points. This imbalance can lead to a lower that expected number of predictions for the faulty category. In order to mitigate this problem, techniques such as sampling or augmenting with other information is typically used [45]. In the case of [45], prediction sampling is used along with code smells to help counteract the imbalance in data, which resulted in improved predictions on the faulty category.

The above two problems make it very clear that the dataset is key to good results just as much as the algorithm being used. To this effect, numerous studies have been performed on the types of data that should be used and the effect the data has on the prediction results. Studies such as [74], [104] and [39] looked to refine the datasets before feeding the data to the classifiers. In this refinement, they looked to typically remove redundant data, normalize the data and then perform feature selection. Feature selection is the process of finding the minimum set of features within the dataset that best describe the data with little to no loss of applicable information. As a result, feature selection algorithms allow to reduce noise in the data thus reducing the probability of a wrong classification [27].

Machine learning algorithms have helped push the fault prediction field forward to the next level quite quickly. However, as has been seen so far, there are still a number of problems with the techniques used: results highly dependent on metrics used, datasets can be imbalanced, excessive noise in datasets and each classifier is able to only detect certain fault trends in the code. In order to counteract some of these problems, researchers have looked into using ensemble learners for predicting faults. Two different studies evaluated the performance of various ensemble learning techniques and both concluded that the ensemble classifier outperformed the individual classifiers [50, 86]. However, Kumar et al. stated that the results appear to be dependent on the set of metrics chosen. They point out that selecting the right subset of code metrics is still important to achieving good results, just as pointed out by Wahono in his literature review [99].

One of the keys to making ensemble learning algorithms useful is to have a variety of classifiers that will not show the same characteristics. With that in mind, [84] created an ensemble learner, using stacking ensembles that embedded a diversity function into the learner. The diversity function gave weights to the individual classifiers in the ensemble learner based on accuracy and diversity compared to the rest of the classifiers. The resulting learner performed better than single classifiers and a few other common ensemble learning techniques.

Another technique that is being researched to overcome the issues with the datasets is to use meta-heuristic and fuzzy logic. The intent is to partially allow the models to be human understandable and to be able to deal with the large biases in the datasets. As such, [93] and [10] look to apply various forms of fuzzy logic along with other known techniques to improve the prediction results. [10] uses a neuro-fuzzy network in conjunction with genetic algorithm to predict faulty modules. Similarly, [93] uses a fuzzy rule generation methodology alongside feature selection that performs better than numerous single classifiers.

The one large concern though is that most fault prediction studies use what can be considered to be pre-groomed datasets. These datasets are stored in open-source repositories like PROMISE [68] and NASA MDP. They have been used in numerous prior studies. The datasets allow for repeatability and comparison studies without having to repeat prior studies on new datasets. The reason they are considered groomed is because most of these datasets have the faults pre-marked within the application and code history by a researcher. Studies such as [95] that have developed frameworks to predict faults tend to use such repositories to evaluate their models too. The advantage as mentioned before is that it does allow for reproducibility and comparison, however the grooming cannot be replicated on larger applications. These repositories typically contain applications with a couple hundred classes, which does raise some concerns when the model have to be run on applications with a few thousand classes in them. Such a large difference in code size and history does not make it a feasible solution to groom the source history to mark faults within the code similar to the groomed datasets. Similarly, the difference in code size can also translate to the need for more extensive tuning of the classifiers for them to be useful on the datasets.

In order to evaluate the fault prediction capability of each of the proposed techniques, studies typically use the same performance measures as those used to evaluate classifiers, which are Precision, Recall, Receiver Operating Characteristic Curve and F-measure amongst others [62]. Along with these measures, this thesis uses one more evaluation technique known as *Precision* @ K. Precision @ K is the percentage of correctly identified faulty classes amongst all the classes marked faulty when the top K classes are selected from the list of classes sorted in descending order of their probability of being fault prone. This method helps to visually understand how well the predictor performs when selecting the top K faulty modules. Having good confidence in the modules rated the highest for faults is crucial, otherwise the predictor might be considered under-performing and thus not useful overall.

2.4 Summary

This chapter has looked at the research that has been conducted so far in the areas of TCP, Tie Breaking for TCP and Fault Prediction. Each of these fields have come a long way but there still exist obstacles to overcome and areas to improve upon.

When it comes to TCP, there is a heavy reliance on simulated faults in the form of hand seeded or mutation injected faults. There is some correlation between real faults and mutation faults when it comes to test cases detecting the faults. However, it appears that the mutation faults might be too simple. As a result, there is the need to run TCP experiments on real faults to gauge their true performance.

Tie breaking in TCP has not garnered enough attention yet even though the current choice of random selection can have a negative impact on fault detection rates. When tie breaking has been taken into account in studies, the procedure still depends on a variation of the primary criterion. There is a need to investigate secondary criterions to use as part of tie breaking.

Finally, when it comes to fault prediction, there are two issues that arise when dealing with supervised learning algorithms. The first is that most studies have been performed on pre-groomed applications or datasets. For initial studies, it is great to use these datasets for comparison and validation. However, there needs to be more work done to address fault prediction in large systems where pre-grooming is not a viable option. The second issue is the classification algorithm selection problem, in that no single classifier can predict on all datasets. Instead, one needs expert knowledge to select and tune an algorithm for use with each dataset. Due to these two problems, there is a need to investigate using non-groomed datasets along with possibility of universal classifiers to be able to predict on large applications.

Chapter 3

The Framework Architecture

In this chapter, the design of the framework to supervise tie breaking within TCP will be presented and discussed. The chapter will start by discussing some of the goals and qualities that dictated the final architecture of the framework. This will be followed by an in-depth discussion of the data flow and the various parts of the framework along with how they help achieve the final goal.

3.1 Framework Goals and Qualities

The primary goal of the framework as stated earlier is to supervise tie breaking in TCP algorithms. One approach to achieving this goal is to just right a whole lot of code with no real structure. The actual code might work and return reasonable results, but could be unusable beyond the study. The more prudent approach is to design the framework for versatility such that it can be re-used for future studies and applications. *With that in mind, the goal is to design a framework that is versatile and self-sufficient for applicability to a variety of large applications with the intention of supervised tie breaking within TCP.*

The qualities that were deemed necessary to achieve the goal are as follows:

- Generic - The framework needs to be able applicable to multiple situations. It should be not be tuned for any particular application or use case.
- Modular - The framework needs to be distinct components that can be swapped out easily for upgrades or modifications during future use.

- Extensible - The framework needs to be easily extendable for future research or applications. This works hand-in-hand with the modular aspect to make the framework more versatile.

Based on these goals and qualities, the framework's architecture is shown in Figure 3.1. The framework is split into two distinct set of components: Fault Predictor and TCP Tie Breaker. As implied by the names, the Fault Predictor components are responsible for predicting faults in the applications modules. The TCP Tie Breaker components are responsible for prioritizing test suites based on coverage information and fault predictions from the first set of components. The workflow of the two sets of components along with the description and responsibilities of each of the components will be presented in the rest of this chapter. The final implementation of the various components in the realization of this framework is left to the implementers discretion.

3.2 Workflow

The workflow for testing a new version of an application under development typically relies on information gained from testing previous versions of the software. Testers use coverage information and/or expert knowledge gained from previous test results to help direct testing effort for the new version. In this way, they try to maximize their fault detection efficiency once the application is code complete.

In a similar manner, the framework shown in Figure 3.1 works to use prior information to guide future actions. To accomplish this, the framework splits the work into two distinct phases. The first phase, called the *training phase*, uses the history of the application to collect software metrics and train the classifier for fault prediction. This effort along with the gathering of coverage information can be conducted during the development cycle of the new version. The second phase, called the *prioritization phase*, takes place after the new version is code complete. During this phase, the framework calculates the fault probability of all existing using the trained classifier. The framework then uses the gathered coverage information and the fault probabilities to prioritize the test suite for the new version. The workflows and the components involved in each phase can be seen in Figure 3.1 where the solid black line represents the training phase and the dashed green line represents the prioritization phase.

Taking a closer look at the data flow, the process starts with the training phase with *Test Suite & Coverage Discoverer*, *Code History Parser* and *Metrics Collector* components

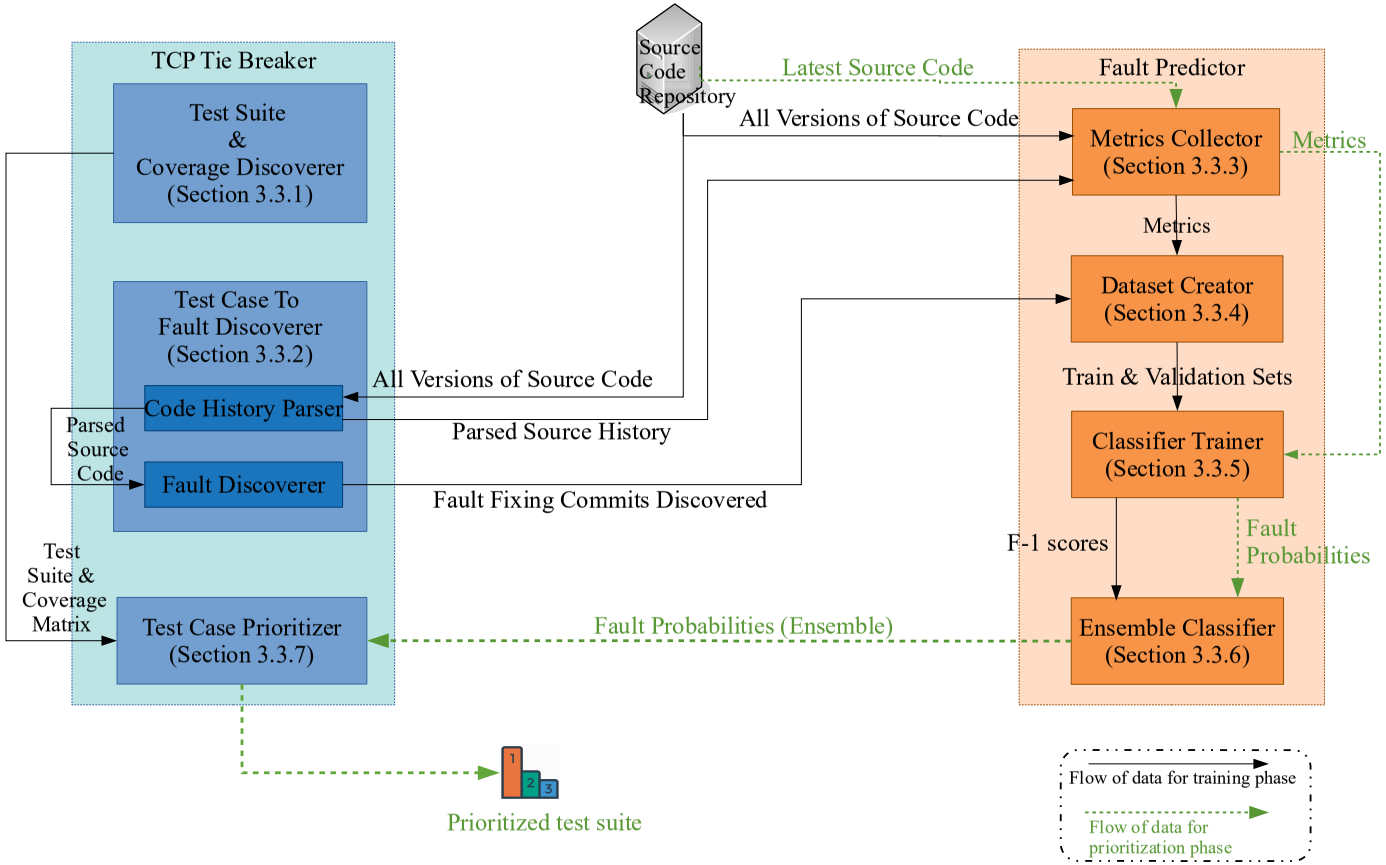

Figure 3.1: Architecture of the Framework

gathering the base information needed by the framework. The base information includes the parsed source history, software metrics and test cases. The *Fault Discoverer* uses the parsed information from *Code History Parser* to identify the faults that existed in the source history. The *Dataset Creator* uses the fault history and collected software metrics to create the training and validation datasets. The *Classifier Trainer* uses the datasets to train and validate the classifiers, whose results are passed onto the *Ensemble Classifier*. At this stage, the training phase is completed and the prioritization phase begins. During the prioritization phase, metrics are collected for all existing classes by the *Metrics Collector* provided to the *Classifier Trainer*. The *Classifier Trainer* has all the trained classifiers predict on all the metrics. These predictions are given to the *Ensemble Classifier* for final fault predictions. These final fault predictions are passed to the *Test Case Prioritizer*, which uses the prediction values along with coverage information to complete the prioritization of the test suite. At this point, the user of the framework will receive their prioritized test suite.

3.3 Details on Components

Based on the data flow, the rest of this chapter will go into details of the various components. Each component's functionality will be explained with regards to their role(s) in each of the phase(s).

3.3.1 Test Suite & Coverage Discoverer

In order to be able to prioritize tests within an application, there are two requirements. The first requirement is to know the set of tests that exist and need prioritization. The second requirement is the criteria by which the prioritization will take place and how it relates to each test. The prioritization criteria used in the framework is the code coverage achieved by each test case at the granularity level of the class.

The Test Suite & Coverage Discoverer component, as its name suggests, identifies all the tests within the application source code and collects all their code coverage information. This component is the only one in the framework that will need to be tailored to each application. The tailoring is necessary due to two main reasons. The first reason being that the test structuring is not always the same between applications. There are some general rules used to organize tests, but nuances exist between applications. The second reason is that there exist a variety of tools to run tests and collect their coverage information. Each

application, based on their technologies and structuring, could use a different toolset to execute their test code. As a result, there is the need for tailoring to each application.

When considering code coverage though, just knowing the tool and how to execute it is not enough. In most cases, the tools are configured to collect overall code coverage. The primary concern typically is to either increase or maintain the coverage when making modifications. In addition, by completing a delta analysis of the coverage before and after a change, it is possible to make sure there has been a gain in the code coverage. TCP on the other hand, needs to know the coverage of each individual test case instead. Due to this gap in general use of the toolsets versus the needs of coverage-based TCP, there is a need to build custom components to gather the needed information. This component is meant to be the custom component that is able to setup the right execution environment to gather and record the needed coverage information. The gathered coverage results are stored as a matrix of test to class coverage. To gather the needed coverage information, the component always uses the latest available code from the source code repository. As an extension, it is possible to have the component keep state and only track the coverage for test cases that have been modified since the last execution. All of this work is completed as part of training phase of the framework.

Once the component has completed its work, the initial information needed to perform coverage-based TCP will have been aggregated. There is still the need to generate the test case to fault detected matrix to be able to complete evaluations using APFD and the fault proneness of modules to supervise the tie breaking within TCP.

3.3.2 Test Case to Fault Discoverer

Evaluating test suites is typically conducted using a measure known as APFD. To be able to use this measure, it is necessary to understand the relationship between test cases and the faults they detect. There are two approaches to gathering this information: intentionally adding the faults and the detecting test cases or identifying the previous faults and the test cases that detect these faults. The approach taken by the Test Case to Fault Discoverer component is to use real faults from the code base along with the test cases that can detect them. In this way, the evaluation is based on real faults.

The component gathers the fault information and the detecting test cases in two steps during the training phase of the framework. Each step is performed by a separate sub-component, each described below.

The **Code History Parser** is the first step in identifying the historical faults within the source application by parsing the source code repository. Changes in source code are

stored in the repository as an atomic set of modifications known as *commits*. Each of these commits are highly detailed with regards to the code modifications performed. They contain the time of the modifications and the author of the modifications amongst all the information available. When these commits are ordered chronologically, it provides a detailed development history of the application.

In accordance with the need to understand the application's history, the Code History Parser walks through each and every commit in the source code repository and stores the relevant details. For each commit, the sub-component extracts the date, commit message, modified files along with type of modification, commit identifier and author information. The type of file modification extracted from the commit details whether the file was copied, renamed, modified, added or deleted as part of the modification. The sub-component stores all the commit history information gathered in a database so that they can be used for future analysis by the components within the Test Case Prioritization Tie Breaker and Fault Prediction.

The **Fault Discoverer** is the second step in identifying the historical faults and the test cases that detect them. This subcomponent uses the parsed history from the Code History Parser to perform its task.

To identify the faults within the source history, the Fault Discoverer analyzes the commit message of each and every commit of the application. In the commit message, it looks for variations of the word 'fix' as an indication that a fault has been fixed in the commit. This was a technique successfully used by [48], [63] and [72] in identifying faults in their studies. Another reason for this approach was that during the analysis of a few large applications along with their source and bug repositories, it was found that there consistent link did not exist between the commits fixing a fault and resolved bug reports. The resulting fault fixing commits identified from this analysis are stored in the database for future use.

On identifying all the faults within the source history, the test cases that were modified or added to detect each of the faults can be established. As discussed earlier, the files modified within each commit have been parsed and stored in the database by the Code History Parser sub-component. Using this pre-parsed information, it is possible to identify the test cases that accompany a fault fix. Based on the test cases identified in each of the fault fixing commits, a matrix of test cases to fault commit is created. This matrix shows the relationship between all existing test cases and the faults they detect. The APFD reports needed for evaluation will be generated using this matrix. The fault information gathered by this sub-component will also be utilized by the Dataset Creator component during fault prediction.

3.3.3 Metrics Collector

Data or information is the basis to learning. A system or individual needs to be able to analyze and understand relevant data to be able to achieve the desired learning. When it comes to classifiers, the relevant data is one that is quantifiable in nature. Classifiers work best on numerical data and thus the code needs to be broken down into meaningful numerical or quantifiable values.

The Metrics Collector is the component responsible for calculating all the necessary software metrics. The component runs in both phases of the framework, training and prioritization. During the training phase, the component is tasked with analyzing each and every code file within each and every commit of the application. The analysis consists of calculating a variety of software metrics and storing them for future analysis. The set of software metrics calculated for fault probability is meant to be determined by the implementer of the framework so as to allow for flexibility where necessary. This flexibility should allow for software metrics more suited to the technologies being used or updating the component for new applicable software metrics. The only requirement during implementation of this component is that there be a few different classes of software metrics and they all be numerical in nature. All the metric values are then stored in the database for use in training.

In the prioritization phase, the component again calculates the software metrics for all the available code files. The difference in this case is that the calculations are done on the latest version of the code. All available code files available in the code repository at the latest commit are analyzed and submitted for prediction to the next component.

3.3.4 Dataset Creator

When learning to distinguish two types of things apart, as humans we need examples of each thing to be able to identify what is different about the two. This comparison allows us to differentiate between the two types when encountered later on. Similarly, classifiers need information about each category so as to be able to learn the unique features of each. For this purpose, it is necessary to generate two datasets for fault prediction, non-faulty and faulty. The datasets need to be as disjoint as possible so as to allow the classifiers the best possible chance of finding the discerning features of the dataset. The job of creating the two datasets is performed by the Dataset Creator component and is executed only during the training phase of the framework. From this point on, the dataset that is made of classes within fault fixing commits will be called the ‘high risk’ dataset and then second set will be called the ‘low risk’ dataset.

The component splits the work of generating the datasets into three tasks. The first task is to extract the software metrics for all the classes in the 'high risk' dataset. This extraction is performed using the fault information collected by the Fault Finder sub-component and the software metrics calculated by the Metrics Collector. Any file that has been modified in a commit that addresses a fault is considered to be a 'high risk' file. The component extracts the software metrics for each of the classes identified in the files that have been modified in the fault fixing commit. When extracting the software metrics, the values that correspond to the last modification of the class before the fault fixing commit in question is used. Extracting the software metrics before the commit allows the classifier to understand the state of the classes before a fix was implemented and thus the state of the class when the fault existed within it. This aligns with the aim of identifying classes within the code base that might be at risk of containing faults. If we take the software metrics after the fix has been applied, we only get the snapshot of the result and not the snapshot of the class with the fault within it. The gathered software metrics are cached in the database, each set being grouped by the fault fixing commit they belong to.

The second task performed is to aggregate the 'low risk' dataset. To gather these values, the files modified between two fault fixing commits are identified. If any of these files have been modified in the second fault fixing commit, they are discarded from this set. In other words, assume there are two fault fixing commits, A and D, with two other commits in between, B and C. If there exists a files within B or C that has been corrected in D, the file will not be included in the 'low risk' dataset. These values are stored in memory until the third task is completed.

The third task performed is the removal of duplicates from the datasets. Faults are an unwanted by-product of development efforts and thus will show up in the code base in time. As a result, a set of software metrics that appears in the 'low risk' dataset will definitely appear in the 'high risk' dataset at some point when the fault is identified and corrected. If left as is, the same set of software metrics will appear once in the 'low risk' dataset and once in the 'high risk' dataset, breaking the rule of disjoint datasets. To rectify this situation, it is necessary to remove these duplications and create two disjoint sets of data. To achieve this, the component compares the software metrics in the 'low risk' dataset to the 'high risk' dataset class by class. The reason for identifying duplicates only for the same class is that multiple classes having the same metric values allow for a stronger learning as the values get reinforced. When duplicates are encountered, they are removed only from the 'low risk' dataset. The collected software metrics are then stored in the database as the 'low risk' dataset along with the class identifiers and commit identifiers just as the 'high risk' dataset are stored. The commit identifier stored in this case is that of fault fixing commit that occurs right after the modification to the class was committed.

Once the two sets are created, they are combined into one set as a list of tuples, $\{(\mathbf{z}_i, y_i)\}_{i=1}^{n}$, where $\mathbf{z}_i$ is the i-th set of software metrics and y_i is the label for the i-th set. During the combining step, the tuples are grouped together by commits so as to create time-slices of history. By storing as time-slices, it is possible to mimic the progression of code and it creates a definite sequence of input for any classifier whose training can be affected by the order of input of the training data. This list of tuples is further broken down into a train and validation set. The split is performed on the number of commits in the dataset and not on the number of data points. This is so as to mimic the progression of code in time-slices and will be addressed during the training of the classifiers in the next sub-section.

3.3.5 Classifier Trainer

The Classifier Trainer, as the name suggests, is the component that performs the training of each individual classifier as part of the training phase. The first is to pre-process the datasets for ingestion by the classifiers. The second step is to train and validate each of the classifiers. The component is also utilized as part of the prioritization phase to get predictions from each of the classifiers.

The pre-processing step manipulates the data as needed before it is fed to the classifiers for training, validation and prediction. By pre-processing data, it can be massaged into the necessary format and dimensions as needed by each classifier. It can also be used to reduce noise and normalize values by techniques such as dimensionality reduction if it is deemed necessary.

To perform this step, the component receives the dataset $\{(\mathbf{z}_i, y_i)\}_{i=1}^{n}$ as prepared by the Dataset Creator. Each tuple's $\mathbf{z}$ set is then manipulated (for example dimensionality reduction techniques can be applied if necessary) and the resulting set is used to train and validate each classifier. If dimensionality reduction is performed on the data, the resulting reduction scheme is stored for performing dimensionality reduction on the data during the prioritization phase too. The component can perform multiple manipulations if needed and build multiple datasets as a result. For example, one dataset could be created by using Principal Component Analysis, while another could use binarization and then Principal Component Analysis.

By allowing for multiple manipulations and multiple datasets, it allows for experts in classification algorithms to analyze and compare the data with regards to each classifier. It also allows for further analysis on the data in a research or tuning situation. One possible comparison, which has been performed in this thesis, is how well the classifiers are able to

clean out noise in the data by themselves. In the case that the classifiers are able to detect different fault sets by just manipulating the data, it allows for larger number of trained classifiers using the same base set of classifiers.

Overall, this approach is meant to allow for a bit more flexibility in the data being ingested by the classifiers. Small algorithm changes can be made without changing the actual data that has been extracted and aggregated into datasets.

Once the data has been pre-processed for ingestion, the framework can start the process of training the classifiers. The Classifier Trainer component is meant to be able to handle any classifier as long as they follow a standard interface to ingest the training, validation and prediction data. Using such an interface, allows for adding, removing or swapping of classifiers very easily for future use. This ability to change the classifiers used allows for greater flexibility in the component. The training and validation is run by the Classifier Trainer separately on each classifier.

The Classifier Trainer is made of r classifiers multiplied by the number of n pre-processors used. The datasets provided at this point have been partitioned into the training and validation sets. As mentioned earlier, the partitioning is performed on the number of commits (in chronological order) in the dataset, rather than the total number of data points that exist. By splitting in this manner, the system has virtually created versions of the data for training and validation, similar to studies using two versions of the same application for their analysis.

Similar to the partitioning, the data is fed to the classifier in batches, where each batch represents a commit. This is meant to mimic source code getting committed to the source code repository in distinct time slices. The main advantages of this approach is that, just as mentioned earlier, any classifier that is susceptible to order of data inputted learns based on chronological order. By training in such a manner, it would be easy to store the state of the classifiers and then add more commits for training as time progresses during a development cycle. Depending on the classifier too, it can allow for learning the changing trends in faults as time progresses and the application gets more complex. At this stage, the training of the classifiers is complete and validation needs to be performed.

The validation of the classifiers begins with the test data being run through each of the classifiers. The test data is fed without the labels one at a time and the resulting predictions are stored. These predictions are compared to the known data point labels and used to calculate the precision and recall for the 'high risk' category. The precision and recall are then used to calculate the F-beta scores using Equation 3.1.

$$F\text{-}beta = (1 + beta^2) * \frac{precision * recall}{beta^2 * precision + recall} \tag{3.1}$$

The F-beta score is the harmonic mean of precision and recall. The beta value changes the weighting of the harmonic mean as can be see from Equation 3.1. When the beta value is 1, precision and recall are given the same weight. When the beta value is below 1, precision is given a higher weight and vice versa. For the framework's use case, F-1 and F-10, beta scores of 1 and 10 respectively, are the specific scores calculated. The F-1 scores are used to evaluate the performance of each of the classifiers along with being an input to the Ensemble Classifier component. The F-10 beta scores are used to assist with the evaluation of each of the classifiers. F-10 scores have not been used in previous studies, however it was found to be a good indicator of recall vs precision when compared to F-1 scores due to its heavy weighting of recall. A low F-1 score could be due to either low precision or low recall or both. When the F-1 score is compared to the F-10 score, it is possible to determine whether precision or recall of the classifier performed poorly. If an F-10 score is lower than the F-1 score, it points to a lower recall value and vice versa.

At this stage of the Fault Prediction components, there exists the set of $r * n$ classifiers that have been trained and validated as part of the training phase of the framework. The classifiers potentially detect different fault trends in the code, which in-turn should make the Ensemble Classifier more versatile.

During this prioritization phase, this component becomes a set of $r * n$ predictors. The component receives the set of classes for which to predict their fault proneness along with their respective software metrics. These software metrics are received directly from the Metrics Collector component with its class name labelled. The received software metrics are pre-processed with the same algorithms used in the training phase, creating n datasets for prediction. These n datasets are then matched with the right classifiers from the $r * n$ set of classifiers for prediction. The resulting predictions are gathered where each prediction is labelled with the class name and the classifier that returned the prediction. These predictions are then given as input to the Ensemble Classifier for the final prediction values on each class.

3.3.6 Ensemble Classifier

As stated earlier, individual classifiers can be biased towards detecting certain types of faults. There also exists the problem of tuning the classifiers based on the data being used, thus necessitating the need for an expert in the field to build a good predictor.

One approach, that is garnering a lot of attention, to overcome these issues is to combine multiple classifiers into an ensemble learner, called the Ensemble Classifier component in the framework. The ensemble learner works on the principle of aggregating information from multiple sources, multiple classifiers in this case, to make a final informed decision. The ensemble learner could use a majority vote, a weighted approach or numerous other techniques to generate the final result. This approach allows for versatility of use while allowing for the ability to detect multiple fault trends in the code.

The Ensemble Classifier component, just like the classifiers, takes part in both phases of the framework. During the training phase, the Ensemble Classifier establishes the schema it will use to generate the final predictions. There are multiple schemas that can be used, some are naïve while others are fairly complex. For efficiency and simplicity, the framework uses a weighted averages technique. The weights of each of the classifiers is calculated using their F-1 scores, as provided by the Classifier Trainer. Using weighted averages allows the framework to take the results of each classifier into consideration while giving importance to those classifiers that performed the best on the validation data.

$$w_j = \frac{F_j}{\sum_{i=1}^{r*n} F_i} \tag{3.2}$$

Equation 3.2 is used to calculate the weight for each of the $r * n$ classifiers. In the equation, F_i represents the F-1 score for the i-th classifier and w_j represents the weight for the j-th classifier. As can be seen from the equation, the weights are highly dependent on how the classifiers perform compared to each other. A classifier that has a relatively higher F-1 score will have a higher weight compared to the other classifiers and similarly a lower F-1 score results in a lower weight. These weights are then stored for applying during the prioritization phase.

$$P_{ensemble}(O_i) = \frac{\sum_{j=1}^{r*n} w_j \times P_j(O_i)}{\sum_{j=1}^{r*n} w_j} \tag{3.3}$$

During the prioritization phase, the Ensemble Classifier receives the predictions from the Classifier Trainer. These predictions are inputted into Equation 3.3 to calculate the final probability value for each class submitted for fault-proneness prediction. In Equation 3.3, O_i is the class for which the fault probability is being calculated, $r * n$ is the number of classifiers, w_j is the weight of j-th classifier, which was calculated during the training phase, and $P_j(O_i)$ is the probability score of j-th classifier for class O_i. The resulting value $P_{ensemble}(O_i)$ is the probability of class i containing a fault.

On completion of all the predictions during the prioritization phase, the Ensemble Classifier aggregates the results into a list of tuples. Each tuple contains the class name and the resulting fault-proneness prediction as a value between 0 and 1. These results are then provided as input to the Test Case Prioritizer to supervise the tie breaking within TCP.

3.3.7 Test Case Prioritizer

The final task of the framework is to prioritize the test suite based on all the information it has gathered so far. The task of prioritizing the test suite is performed by the Test Case Prioritizer component as part of the prioritization phase of the framework. The set of tests to prioritize and their code coverages have been generated by the other components of the TCP Tie Breaker. The Fault Predictor components, described in Section 3.3.2, will provide the fault-proneness information to supervise the tie breaking.

To create the prioritized test suite, the Test Case Prioritizer uses AT. As mentioned in Section 2.1, AT is one of the more common techniques used as a foundation for more complex prioritization methodologies. To perform AT, the component uses the test case to code coverage matrix generated by the Test Suite and Coverage Discoverer component. As noted earlier, the technique does suffer from the problem of ties occurring fairly frequently during the selection of the next test case.

In order to break the ties, the component applies three different tie breaking methodologies, two new supervised tie breaking methodologies being proposed by this thesis and a base tie breaking as a comparison. Due to there being three tie breaking methodologies, three prioritized test suites will be generated by the component. The three methodologies are as follows:

1. Binary (ATB) - This approach is to add the fault probability values of every class covered by the test case. The more the classes and the higher the fault prediction values, the larger the sum. The test case with the largest sum gets selected as the next test case to run in the test suite. The methodology can be represented with the following equation:

$$S_b(T_i) = \sum_{j=1}^{o} \text{sgn}(\mathbf{C}_{ij}) \times P_{ensemble}(O_j) \tag{3.4}$$

   where sgn(.) denotes the sign function, C_{ij} denotes the coverage achieved by the ith test case on jth class, $P_{ensemble}(O_j)$ denotes the fault probability of the jth class and $S(T_i)$ denotes the score of the ith test case.

2. Percentage Coverage (ATP) - This approach multiplies the percentage of code of a class covered by a test by the fault probability value of the class and then sums up all the values. This could be considered a weighted approach to the selection. If a large number of classes covered by the test case have high coverage and high fault probability values, the test case has a higher chance of being prioritized next in the test suite. The methodology can be represented with the following equation:

$$S_p(T_i) = \sum_{j=1}^{o} \mathbf{C}_{ij} \times P_{ensemble}(O_j), \tag{3.5}$$

where C_{ij} denotes the coverage achieved by the ith test case on jth class, $P_{ensemble}(O_j)$ denotes the fault probability of the jth class and $S(T_i)$ denotes the score of the ith test case.

3. Random - This approach randomly selects a test case from the set of test cases that have been tied in coverage. This is the method commonly used when breaking ties at present.

The first two tie breaking methodologies, Binary and Percentage Coverage, are the supervised techniques being evaluated in this thesis. The last methodology, Random, will be used as the control for comparison. As stated earlier, random tie breaking is considered a lower bound method.

The component also takes the test to fault matrix as input to build APFD reports for evaluating the prioritized test suites. During the generation of the APFD report, only the first test case that detects a faults is attributed with the found fault. All other test cases that detect the same fault disregard the fault in question. In other words, if test cases A, B and C all find the same fault and B is the first test case to run, A and C are deemed to not have detected the fault when they run later in the test suite.

The Test Case Prioritizer is the last component to run in the complete framework. It executes as part of the prioritization phase and its output is returned to the user. As stated earlier, the component returns the prioritized test suites and the respective APFD reports. The returned prioritized test suites can be used as input to a test runner and the APFD reports can be used for evaluation purposes.

3.4 Summary

In this chapter, the goals, qualities and architecture of the framework were discussed. The framework is meant to be versatile and self-sufficient so as to be useful in a variety of uses

cases for use as a supervised tie breaker within TCP.

To achieve this goal, the framework is able to detect and gather coverage information for all the tests within the application. In parallel, the framework is able to parse source code history, generate software metrics on all changes and detect faults that have existed in the past. Using the software metrics and fault information, it is able to train an ensemble learner to predict the fault-proneness of classes within the application. The fault-proneness of classes is then used to supervise the breaking of ties when they occur in coverage-based TCP. Finally, the framework returns a set of prioritized test suites for evaluation and use.

Chapter 4

Experimental Setup and Empirical Studies

In the previous chapter, a framework was designed and two tie breaking schemas were proposed to help solve the problem of ties in TCP. Both of them now need to be evaluated to make sure that they actually solve the problem they were designed to solve. In order to be able to perform the evaluation, it is necessary to implement the framework into a usable tool and execute it against some case studies to produce prioritized test suites and fault predictions. These returned prioritized test suites and fault predictions can be used in the evaluation of the framework.

In the rest of this chapter, the experimental setup, the case studies and the obtained results will be discussed. The discussion will include details about libraries and tools used to implement the framework. Details about the selection process for case studies and pertinent details about them will be presented. Finally, the results obtained by running them through the framework will be discussed in relation to the research questions posed in Section 1.2.

4.1 Experimental Setup

In this section, the various details about the experimental setup will be discussed. The languages and libraries used, and concrete details about software metrics and classifiers will be presented.

4.1.1 Languages and Tools

The framework has been designed to be tooling and language agnostic. This was done with the aim for flexibility in implementation and for ease of use for future research. The implementation of the framework completed in this thesis depicts some of the intended flexibility by using readily available tools and libraries.

The framework was implemented in Python 2.7. Python, especially Python 2.7, has been the language of choice for numerous studies [60,91,96]. The ease of use of the language along with its vast set of research focused libraries have made it a popular choice for use in research studies. Due to this popularity, there exist numerous libraries for Python for machine learning applications and statistical analysis. Most of these libraries are open source and thus easily available for use. The database used for storage of all the parsed and intermediary information was MariaDB [3]. It is an open source SQL database that has been forked from MySQL.

Two external tools were integrated into the framework to perform source code analysis and source repository analysis. These tools were CVSAnaly, used as the Code History Parser (Section 3.3.2), and Understand™by SciTools, used as the Metrics Collector (Section 3.3.3). CVSAnaly is an open source tool written in Python 2.7 to parse the commit history within source repositories and has been used in numerous data mining studies [24]. It supports the parsing of a number of source code repositories, understanding how to step through each commit within the various types of repositories. It is also capable of extracting all the necessary information, as detailed in Section 3.3.2, from each of the commits within the source code repository.

Understand™by SciTools is a proprietary tool that is capable of calculating a large variety of software metrics. The tool supports the calculation of software metrics on numerous languages thus opening up the ability to use the same implementation against a variety of applications regardless of their language. When integrated with CVSAnaly, it is able to extract software metrics on every file in every commit. CVSAnaly has the capability of adding custom analysis on every commit as it parses a source code repository. This custom analysis is where CVSAnaly provides Understand™the files modified in the commit in order to complete the software metrics calculation. On completion of the software metrics calculation, Understand™returns the values to CVSAnaly so that the commit values and software metrics can be stored together in the database.

4.1.2 Software Metrics

As mentioned in the previous section, the use of Understand™by SciTools allows for a large number of software metrics to be gathered for each application. Every file modified within every commit of the application was analyzed by the tool resulting in a set of software metrics. The software metrics were calculated for each class stored in each of the files. These software metrics, when aggregated, form the $\mathbf{z}_i$ vectors which are used by the framework to train and predict the fault proneness of classes. A total of 94 different software metric values were calculated on each class in every commit. Most of the software metrics used belong to known suites such as Halstead, McCabe and Chidamber & Kemerer. A few extra software metrics were also calculated to represent details of the commit history of each of the classes, aptly named historical metrics. The historical metrics suite consisted of the following values:

1. Number of times a class has been modified.
2. Number of distinct authors that have modified the class.
3. Number of classes that have been committed alongside the class being studied.
4. Average number of classes that have been committed alongside the class being studied.

Each of the values in the historical metrics suite were calculated in two different ways. The first calculation was performed from the start of time to form a rolling history of the class. The second calculation was performed from the last time the class had a fault fixed within it. If the class did not have any faults fixed before the commit being analyzed, this value was calculated from the start of time. The one historical metric that does not follow the same calculation as the rest is metric 4. This metric calculates an average value instead of just a count. The reason for this historical metric being added was that during large architectural changes or addition of large features, the number of classes committed together become fairly large. These large commits tend to skew the values for the classifier and do not give a good representation of the status of a class to the classifier. By creating an average, it is possible to see how the class was modified along with other classes over time, rather than just one instance dominating the complete dataset. Historical data, similar to those outlined above, have shown correlation to the existence of faults in various studies, making it a prudent choice to include them in the datasets [26, 97].

Table 4.1 gives a break down of the various metrics suits gathered by the framework along with details of how many metrics come from each suite. A more detailed list of metrics used in the framework can be found in Appendix A.

Table 4.1: Metrics Category

Metric Suite	Description	# Metrics
Count	A set of metrics that count various aspects such as the number of lines of code and comments.	27
Halstead	A set of metrics that describe operators and operations in the code along with effort values [31].	9
History	A set of metrics that depict the history of the classes. These include the number of commits, the number of authors and the number of co-commits.	8
McCabe	A set of complexity metrics [65].	21
Chidamber & Kemerer	A set of metrics describing objects in the code, including details on inheritance, coupling of classes and number of methods and variables [12].	29

4.1.3 Test Discovery & Execution

As discussed in Section 3.3.1, the Test Suite and Coverage Discoverer component needs to be tailored to the application being tested. The component needs to know the organization of the code, specifically the test cases, in order to be able to identify all the existing test cases. It also needs to know how to execute the cases and gather the coverage information.

The implementation of these components were completed as independent scripts written in Python 2.7. Since the case studies, presented in Section 4.2, used Jacoco and Cobertura for their coverage reports, the scripts were developed to be able to ingest their respective coverage reports. The scripts ran the tests one at a time and recorded the coverage after each individual run. All results were tallied up into a matrix and then stored as pickle files for later use.

The component was built as scripts for simplicity, parallelizability and ease of use. That being said, in a more permanent solution, it could be built into the framework itself and be made a bit more efficient.

One aspect to note is that each test case for which the coverage has been recorded consists of the full test class. This means that the test case actually consists of multiple tests within it. Similarly, the coverage was recorded for the complete class and not for individual functions in the class. This relates to the software metrics granularity being at the class level and not at the granularity of individual functions within the class.

4.1.4 Classifiers

Machine learning is the area of study of the set of algorithms that allow a computer to simulate human learning with the attempt of making computers intelligent [101]. Amongst the variety of machine learning algorithms that exist, there is a set that are used for predictions. This set of algorithms are typically referred to as classifiers or predictors.

A large variety of classifiers have been developed over the years to help solve numerous problems. They typically fall into one of two categories: supervised learning and unsupervised learning. Supervised learning algorithms are the set of algorithms whereby the machine learning algorithm is fed a set of data points along with pre-labelled information that allow it to learn the differences. Unsupervised learning algorithms do not use any form of labelling to learn about the data, instead they typically aim to create clusters of similarity to be able to classify data [5, 14]. These learning algorithms also typically need an expert to label the clusters for the prediction phase. Given that the aim is for the framework to be self-sufficient, classifiers that need human input to complete predictions is not advisable. This requirement directs us towards using a supervised algorithm along with the fact that the framework is already producing labelled data.

The supervised classifiers used by the framework all come from the library scikit-learn [83]. The library contains a large variety of classifiers belonging to both, supervised and unsupervised, categories. It is an open source library that supports multiple versions of Python and is updated on a regular basis. The library has also been known to be used in prior fault prediction studies [46]. From this library, four classifiers were chosen to be used in the framework: K-Nearest Neighbours [84], Naïve Bayes [38], Bernoulli Bayes [89] and Multinomial Bayes [52].

K-Nearest Neighbours is an algorithm that is based on simple Euclidean Distance of the plotted data points. During training, the data points are plotted in an n dimension space, where n is the number of features in the dataset. Each data point that is plotted has its respective label attached to it too. When predicting on a new piece of data, the algorithm finds the k nearest neighbours to the newly plotted data based on Euclidean Distance, retrieves the labels for these k data points and returns the majority label as the prediction [84]. The algorithm is fairly simple to use and efficient to use when it comes to training and predicting results.

A more mathematical family of classifiers is the set of Bayes classifiers. The Bayes family of classifiers works on the principle of independence between features. This family of classifiers uses a probability model built from the data fed to it during training [15]. The three versions used in the framework manipulate the data in different manners before

building the probability models. Naïve Bayes builds the probability model from the data fed during training without any pre-processing, thus naming it the Naïve algorithm. In Bernoulli Bayes, the inputted data is converted to a binary set and then fed to the probability model [52]. Binarization is done based on some threshold value for each feature set. Multinomial Bayes assumes the inputted data has a multinomial distribution and implements a set of probability equations within the model to support such a distribution [52].

All four of the above mentioned classifiers have been used in previous fault prediction and fault localization studies to great effect. The caveat being a few studies concluded that the classifiers do depend on the data inputted in them and thus could be detecting only certain faults or types of faults [99]. Similarly, the *no free lunch theorem* [103] states that there is no universal learner that will be able to perform well on all tasks. To mitigate the issue of classifier performance dependence on data, the framework uses an ensemble learner for final predictions as mentioned in Section 3.3.6

4.1.5 Training Classifiers

As noted in the previous section, the classifiers being used are from the supervised learning category. Training these supervised learning algorithms requires pre-labelled datasets that are typically disjoint. Using the datasets, they are able to identify features within that distinguish the categories from each other. Once the training is completed, it is imperative to verify that the classifiers can in fact label previously unseen data correctly. This verification, also known as the validation step, is conducted using a dataset with known labels whereby the data is fed to classifier without labels so as to get predictions back from it. These predictions are compared to the known labels and used to evaluate the accuracy of the classifier.

Performing these training and validation steps need multiple datasets. One way of producing these datasets is to split the known data into the needed training and validation sets based on percentage and/or criteria. The split implemented in the framework was 95% training data and 5% validation data. The reasoning for such a large training set was to allow the classifier the maximum possible dataset for learning the discerning features while leaving a statistically significant amount of data for validation. Due to the large history of the applications being analyzed, the 5% testing set was deemed to have enough data points within it to be of statistical significance.

4.1.6 Dimensionality Reduction

Almost all data, specially if gathered from a real source like code metrics, contain some amount of noise or fuzziness. This translates to the various labels or categories within the datasets not always being disjoint. Datasets that are not disjoint tend to make it more difficult for classifiers to learn the discerning features of each category. If the dataset contains too much noise, it is very possible that the classifiers will not be able to identify the discerning features at all and return erroneous categories during predictions. To tackle this problem of non-disjoint or noisy datasets, dimensionality reduction was introduced as one of the pre-processing steps prior to feeding the data to the classifiers.

Dimensionality reduction is a method used to reduce the number of features in a dataset as an attempt to clean as much noise as possible from the dataset [93]. It attempts to find the set of significant features that maximize the data variance and lowers the number of dimensions. Significant features could be single features from the original dataset or could be a combination of features to create a new feature. Given that the framework implementation contains 94 distinct metrics, there is a high possibility of noise within the dataset. Removing said noise using dimensionality reduction would be prudent and helpful in generating reliable results.

Principal Component Analysis is a very common dimensionality reduction technique. It has been effectively used in a number fault prediction studies [9]. Principal Component Analysis works to find the set of linearly uncorrelated features, principal components, that try to maximize variance [28]. It attempts to find features with variances in descending order whereby every new feature identified is orthogonal to every other feature identified before it. In practical use cases, a covariance matrix is constructed using the dataset. Using this covariance matrix, eigenvectors and eigenvalues are computed. The eigenvalues are then summed together until they reach a pre-determined threshold. On reaching the threshold, the number of eigenvalues summed together is the number of features in the new datasets. In the case of the framework, the pre-determined threshold was set to 95%.

Using the dataset created from Principal Component Analysis would also allow for comparison of classifier results between raw and dimensionality reduced datasets. If there is a huge swing between the two due to noise, the raw dataset could be discarded at later stages if desired.

4.2 Case Studies

The framework in Chapter 3 has been developed with the intention of being used against larger applications. The framework is intended to be able to complete all tasks without the need of human supervision. In order to evaluate the framework under these circumstances, there were five requirements in the selection of applications to be used as cases studies. These requirements are as follows:

1. The applications need to have an extensive test suite that can be used as a regression test suite. If a test suite does not exist, there is no way to evaluate a Test Case Prioritizing algorithm.

2. The applications need to be complex in nature and the overall size needs to be large in terms of number of classes and development history. A framework cannot be deemed self-sufficient if it cannot process large amounts of data without human intervention.

3. The applications need to be open source. By having open source applications, it allows for easy access to the code history, thus allowing for a code base that can be independently analyzed. It also allows for repeatability studies and further iterations using the same applications.

4. The applications need to belong to different domains. This was to evaluate whether the framework can easily adapt to different types of applications and thus satisfy the goal of the framework being versatile.

5. The applications need to be in active development. By using applications that are in active development, further improvements to the framework can be evaluated against the new and evolved versions of the software.

Based on these requirements, the applications Cassandra and Tomcat were selected as the case studies. Both applications have been in development for a number of years and have a rich code history. The applications have also had numerous developers commit code, thus allowing for the possibility of a wide range of technical skills affecting the applications. Both applications have readily available source code repositories in public git repositories. These public git repositories were used for data collection and analysis. They also have fairly large test suites that can be used for TCP. The selection of these applications also satisfies the requirement of having different domains: a database and a web server . The details of each of the applications: languages, overall size, number of commits and number of test cases can be found in Table 4.1.

Figure 4.1: Statistics and Information about Selected Large Scale Applications

Name	Language(s)	# Commits	# Classes	# Fixes	# Test Cases
Cassandra [1]	java, python	24076	13330	3222	389
Tomcat [2]	java	21440	6130	6183	533

Table 4.2 presents details on the total number of data points collected from classes within a fault fixing commit, the faulty dataset, vs the number of data points from classes not within a fault fixing commit, the non-faulty dataset. The table also provides a breakdown of the number of data points within the test set for each label. Observing the data, it can be noted that the ratio of total faulty to total non-faulty data points reduces with the age of the application. Cassandra, a younger but very active application, has an approximate ratio of 1:4 for faulty:non-faulty data points, whereas Tomcat has an approximate ratio of 1:2.

Table 4.2: Large Scale Application Data Details

Name	Faulty		Non-Faulty	
	Total	Test	Total	Test
Cassandra	4938	234	22438	1049
Tomcat	8536	364	12223	945

The details about the data points and how many of each type exist in the test set are a good litmus test for verifying the classifiers later on in the experiment. A precision higher than the ratio of faulty to non-faulty data points would be a good first check of the capability of a classifier as the classifier would be performing better than a random classifier.

The next steps are to evaluate the framework with respect to these two applications, from now on referred to as the case studies. The analysis will first take a look at the performance of the supervised tie breaker and how it compares to AT. Following this, the analysis will look at the Fault Prediction components and their performance at the individual classifier and ensemble learner levels.

4.3 Obtained Results - Supervised Tie Breaking

Most studies so far have used case studies with mutation faults injected or hand seeded faults [19]. These studies have shown a large number of ties occurring during AT. This

thesis on the other hand looks to use the real faults available in large systems as its data sets. As such, it is imperative to understand the frequency with which ties occur in the selected case studies.

Table 4.3: Number of test cases, faults and details of ties within the case studies

Name	**# Total Test Cases**	**# Fault Identifying Test Cases**	**# Faults**	**# AT ties**	**Avg. # test cases in tie**
Cassandra	389	155	124	57	9.5
Tomcat	533	152	151	58	9.9

Table 4.3 provides the necessary information for the evaluation: total number of test cases, number of test cases that have detected faults so far, number of faults, number of ties, and also average number of candidates involved in each tie. The data provided in the table pertains to the data in the test set used for TCP. Details about the split into train and test data set are provided later on in this section. Glancing at the data, it can be noted that ties occur in about a third of the iterations that take place in AT. At the same time, there is an average of 10 classes per tie, which is of high significance when deciding which test case to select next. Leaving such a large decision to chance is not prudent and necessitates an informed tie breaking mechanism. One point to note, the test cases that have been marked as fault detecting were the test cases modified during a fault fixing commit. It was assumed that developers either added test case(s) or modified existing the test case(s) to detect the reintroduction of the fault in future development efforts. All other non-fault identifying test cases have been ignored as part of the evaluation.

To be able to evaluate the tie breaking procedures, there is a need to create datasets for the purposes of training and testing the algorithms. This would be similar to training with one version of a software and testing with the next version of the software. To build the training and testing sets, the data is split into 80% training and 20% testing. The split is performed by ordering the commits in chronological order and splitting based on the number of commits, similar to that of the fault predictor split. This is meant to mimic using two sequential versions of a software, one for training and the second for testing. The fault prediction components are fed the training set as the data to train and validate upon. The resulting trained Ensemble Classifier is used to predict the fault-proneness of covered classes and thus supervise the tie breaking.

Upon running the complete framework on the two case studies, each case study produces three prioritized test suites. To evaluate the three test suites, APFD graphs are generated

for each along with the calculation of their APFD values. The resulting APFD graphs are presented in Figure 4.2 and the APFD values in Table 4.4. The APFD graphs also include a theoretical optimal solution and the minimal case produced from random tie breaking for comparison. If we compare the APFD values of the three prioritized test suites to the optimal solution, there is a significant improvement possible. It is however just a theoretical improvement as there is no easy way to knowing before hand how many faults each test case might find.

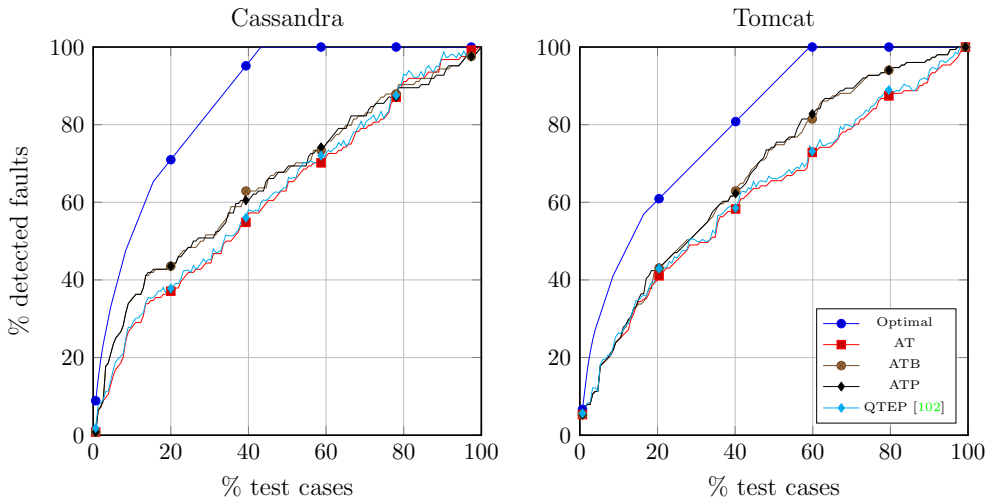

Figure 4.2: APFD of three techniques, additional technique (AT), additional technique with binary tie breaker (ATB), additional technique with percentage tie breaker (ATP)

Table 4.4: APFD values

Mechanism	**Cassandra**	**Tomcat**
AT	61.6%	62.0%
ATB	**65.6%**	67.2%
ATP	65.4%	**67.5%**
QTEP [102]	62.1%	62.5%

As stated earlier, AT uses the code coverage of the test cases to decide which test case should be run next in the test suite. Due to the nature of the technique, the selection of a test case during any iteration of the process affects the remaining selections. This translates to tie breaking procedures being very important as it could lead to a swing in

results as can be seen in 4.2. It can be noted in both case studies that a tie occurs fairly early in the process and from that point on the divergence can be significant.

RQ1 How do tie-breaking schemas that use fault–proneness compare to AT TCP?

From Table 4.4 and Figure 4.2, it can be observed that there is definitely significant improvement in the APFD results. The AT results presented are among the worst case scenarios possible. The two tie breaking schemas consistently outperform AT with improvements of up to 5%. The question does arise of why not apply the fault prediction rates to every iteration of AT. This was the basis of the study conducted by Wang et al [102]. When the same procedure was applied using the fault predictions from the Ensemble Classifier, the results showed only a marginal improvement compared to AT, as can be seen in Figure 4.2 and Table 4.4. This could be due to the fact that fault proneness predicts whether a fault could exist in the class and not the possible number of faults within the class. Thus a class that has one fault or ten faults could have the same fault probability. There also appears to be a higher correlation between coverage and number of faults detected, which all points to the fact that relying on just the coverage is sufficient for the normal iterations of AT.

One of the more important aspects of the experimental setup to note is that unlike similar studies [32, 102], real faults are used instead of mutation faults. The problem associated with mutation faults is that they often follow uniform distribution across the code whereas real faults tend have a non-uniform distribution across the code [23]. According to the Pareto Principle, 80% of faults are found in 20% of the code [29]. This is perhaps one of the main reasons for there being no consensus on whether there is a high correlation between mutation faults and real faults. Some studies have provided evidence for the presence of a weak correlation [44], while others have rejected this hypothesis [77]. Nevertheless, by using real faults the threat of adopting non-representative mutation faults has been eliminated. As such, using real faults places a higher significance on the results obtained from the two proposed tie breaking schemas.

Another advantage of the tie breaking schemas is the repeatability of the results and the ability to positively affect the prioritization early in the process. The affect early in the process is fairly evident from Figure 4.4, whereby the ties occur within the first 10% of the test suite and the resulting divergence in fault detection continues almost till the end of the test suite. It is also of significant importance to note that the complete process has been completed using only the source code history and no human intervention to assist the framework.

Based on the analysis completed of the test suites generated, it can be concluded with

a high level of confidence that the two schemas for tie breaking have a significant positive impact on test case prioritization. Using fault probabilities of the classes as part of tie breaking has helped stabilize the results of Additive Technique.

4.4 Obtained Results - Fault Prediction

The proposed supervised tie breaking schema has shown a significant improvement over AT. The APFD values were higher and more consistent. The supervision though comes from fault predictor models. As such, it is prudent to evaluate the models and evaluate if there is a correlation between the performance of a model and the resulting APFD Value. The rest of this section will evaluate the Fault Prediction components, namely the individual classifiers and the Ensemble Classifier.

4.4.1 Individual Classifier Results

Ensemble learners typically work by merging the results of individual classifiers to produce a more reliable result. This translates to the fact that the ensemble learner's performance depends on the performance of the individual classifiers. If a small percentage of the individual classifiers do not perform well, it should not affect the results of the ensemble learner. However, if all the classifiers underperform, the resulting ensemble learner will most likely underperform too. With these facts in mind, the first evaluation is conducted on the individual classifiers. The performance of the classifiers need to be verified such that most of them can classify the data reasonably well. The evaluation is completed using three different methods: F-beta scores, the Precision-Recall curve and Precision @ K curve.

Starting with the F-beta scores, table 4.5 provides a breakdown of the F-beta scores for each classifier with regards to each case study. One point to note in the results, any classifier that uses the PCA pre-processed dataset has the suffix 'PCA' added to their name. The results for a random classifier is also included so as to be a benchmark for comparison. The random classifier is meant to represent a coin toss decision and depict how a classifier would perform if it were to just guess instead. For the random classifier, a value between 0 and 1 was randomly selected for each data point and if the value was above 0.5 it was deemed to be a faulty class. On inspection of the values, it can be noted that for the most part, with regards to the F-1 scores, the individual classifiers do perform better than the random classifier. However, when isolating Cassandra results, Naïve Bayes

Table 4.5: Classification Results

Classifier Name	**Cassandra**		**Tomcat**	
	F-1 Score	F-10 Score	F-1 Score	F-10 Score
Bernoulli Bayes	0.227758	0.137843	0.461538	0.732946
Bernoulli Bayes PCA	0.227758	0.137843	0.461538	0.732946
KNN	0.144828	0.090425	0.459259	0.426439
KNN PCA	0.14433	0.090421	0.464497	0.43193
Multinomial Bayes	0.419118	0.485618	**0.632219**	**0.851147**
Multinomial Bayes PCA	0.419118	0.485618	**0.632219**	**0.851147**
Naïve Bayes	**0.44898**	**0.697251**	0.38141	0.327851
Naïve Bayes PCA	0.44226	0.385611	0.331288	0.297318
Random	0.270023	0.495757	0.363636	0.50162

and Multinomial Bayes are the only classifiers that have performed better than the random classifier. The main reason is that a random classifier often achieves high recall rate at low precision which is not often useful in practice (one can always achieve the highest recall by labelling all classes as faulty). As mentioned earlier, this is not a great concern as majority of the classifiers are performing reasonably well, thus giving the ensemble learner data that it can perform reasonably well with too. Inspecting the F-10 scores, they do not consistently outperform the random classifier, especially for the classifiers that underperform in the F-1 scores when compared to the random classifier. Since the F-10 scores are weighted more heavily towards the recall of the classifier, it shows that the recall of the classifiers are comparatively low. Having a low recall rate is not necessarily a bad property of a classifier, but it does mean that there are faulty classes being missed. The F-beta scores are a good first litmus test to make sure the classifiers are able to classify our data points reasonably well, however, there is a need to analyze the results further.

Since the aim is to augment the TCP algorithm with fault information to be able to break ties, the classes that are at highest risk of faults need to be identified correctly and early. This translates to the need to verify the precision of the classifiers when it comes to those classes that have been given a high probability of containing faults. To this effect, further analysis will be conducted using the Precision-Recall curve and Precision @ K curve.

In Figure 4.3, the Precision-Recall curves are presented for the various classifiers, including the random classifier. These curves depict how the precision and recall would change as the probability threshold of the classifiers is changed with regards to the high risk category. Taking a deep dive into the classifier results, the precision of most of the

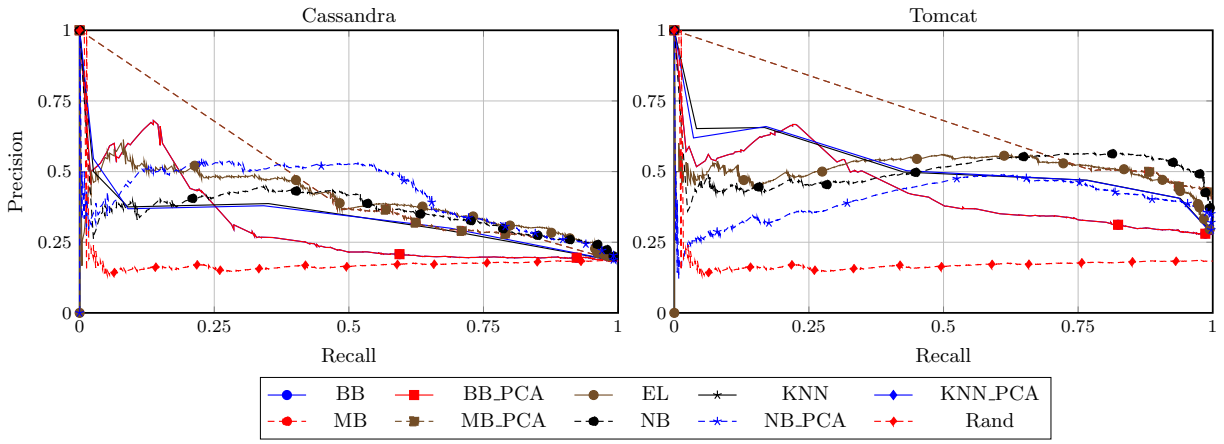

Figure 4.3: Precision-Recall curves for Cassandra and Tomcat.

classifiers are higher than the random classifier for almost all recall levels. Naïve Bayes PCA is the exception where the precision does drop for low recall values, however the precision does improve fairly quickly and outperforms the random classifier. This first analysis validates that the classifiers can at least discriminate between the two labels better than random guessing.

From Figure 4.3, we can make a few more observations about the results. The first point to note is that other than Naïve Bayes, the classifiers have almost identical results whether they were given the copy dataset or the PCA transformed dataset. From this, it can be inferred that the classifiers were able to select the right set of features without the need of feature selection. On the other hand, Naïve Bayes appears to be affected by extra features and/or noise in its dataset. This is not of large concern though as it is known that Naïve Bayes is a fairly simplistic classifier compared to most others and might need tuning to return better results. Having the dual datasets though have strengthened the results of all classifiers that return the exact same result.

The final method in which the results will be analyzed is by using *Precision* @ K retrieval metric. This approach analyzes the top K returned results when the results are ordered in descending order by fault probability. This analysis methodology lends itself well to analyzing the results as if they were to be ranked in priority of testing and allows to build confidence for a user as they investigate the first few results returned by the classifier. It also lends itself well to making sure that the classes with highest probability of containing faults are actually accurate and thus applicable to tie breaking in TCP.

To build the graph, precision is calculated on the top K classes ordered by fault prob-

ability in descending order. By increasing the value of K to the number of classes in the application, a curve can be created that depicts the precision for each value of K. It is a good way to visualize the performance of the classifiers as one iterates through the ordered results and compares to the true labels of the data points.

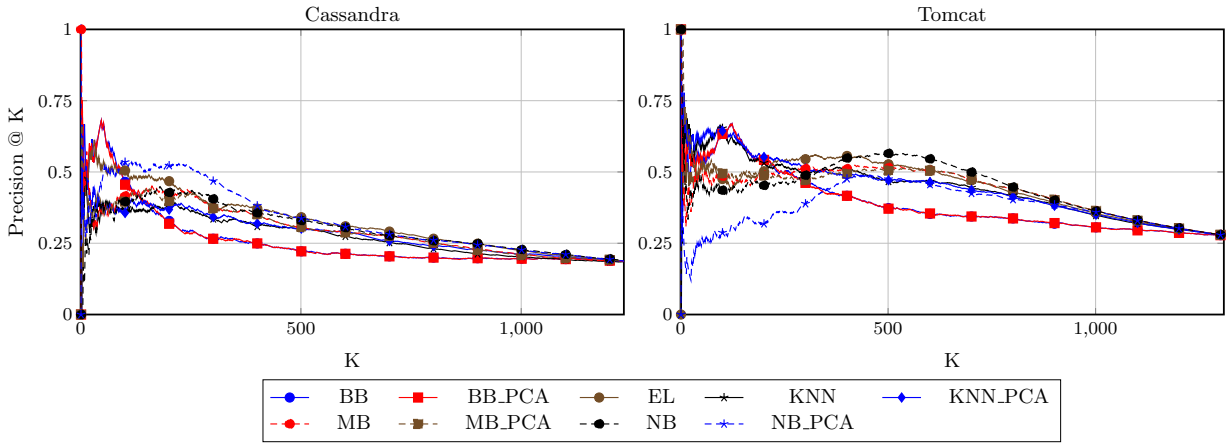

Figure 4.4: Precision @ K for Cassandra and Tomcat.

The precision results of retrieving the top K classes are depicted in Figure 4.4. It can be noted that most of the classifiers have fairly high precision in the first few values of K, however the results do have a large variance for these smaller values of K. The large variance refers to the fact that precision tends to have a swing of 10% to 15% for these smaller values of K. As the value of K increases, the variance of the precision reduces and all the results do converge, as expected, to the ratio of faulty to non-faulty classes. There is some concern that Naïve Bayes PCA is having trouble classifying the probabilities for Tomcat, however it is not significant as it is one out of eight classifiers showing a different behaviour.

Overall, looking back at the various analysis completed, it can be concluded that the individual classifier are able to discern the features to label the classes as high risk or low risk with fairly good precision. When focusing the analysis on the classes that have been assigned the highest probability of containing faults, the precision values increase for most of the classifiers, thus building further confidence in the classifiers and their ability to categorize the data. There have been instances of classifiers performing poorly in certain situations, but given that majority of the classifiers perform well at all times, these one off instances can be ignored. The Ensemble Classifier should be able to discount these results or give them very little weight.

4.4.2 Ensemble Classifier

The last section performed an in-depth evaluation of the individual classifiers. The evaluation concluded that the classifiers have fairly good precision and thus their results are acceptable to use as part of the Ensemble Classifier. In this chapter, the Ensemble Classifier will be evaluated. As mentioned earlier, the Ensemble Classifier employs a weighted average to calculate its final predictions. The evaluation of the Ensemble Classifier will employ the three same methodologies that were used when evaluating the individual classifiers.

The F-beta scores of the Ensemble Classifier can be found in Table 4.6 and Figure 4.3 presents the Ensemble Classifier's Precision-Recall curve compared to the individual classifiers. From Table 4.6, we can note that the F-beta scores are fairly close to the best values in Table 4.5. This is expected as the weighted average based on F-1 scores will cause the resulting Ensemble Classifier to have results close to the highest performing classifier but not be able to match it necessarily. Further analysis shows that the F-10 scores are reasonably high, pointing to good recall performance of the Ensemble Classifier. From Figure 4.3, we can observe that the variance of the precision of the Ensemble Classifier over all values of recall is a fair bit smaller than the individual classifiers. This is expected as the Ensemble Classifier is able to remove any existing bias from individual classifiers and detect a larger variety of fault types at the same time.

Table 4.6: Ensemble Learning Classification Results

Case Study	**F-1 Score**	**F-10 Score**
Cassandra	0.43787	0.473577
Tomcat	0.626068	0.800417

Continuing the analysis of the results, when comparing the *Precision* @ K, a similar trend can be found. The precision for the Ensemble Classifier tends to the higher values shown by the individual classifiers and in some instances of K the Ensemble Classifier matches the best result from all the individual classifiers. This again confirms that the Ensemble Classifier is negating the bias of the various classifiers without compromising the final result. There is a high likelihood of some of the faults being missed due to the ensembling of results, however this is expected and acceptable given the flexibility of use that the Ensemble Classifier provides.

RQ2 How viable is it to use a completely automated tool to predict fault-proneness??

At first glance at the data, it is a little fuzzy as to whether the set of Fault Prediction

components have achieved their goal of identifying faulty classes in the code base. Revisiting the evaluation completed earlier, the individual classifiers showed mixed results for F-1 when compared to the random classifier. When evaluated on Tomcat, the F-1 results were almost always higher than the random classifier, however the same could not be said when evaluated on Cassandra. The low F-1 scores were deemed to be due to low recall after analyzing the F-10 scores. On the in-depth analysis, the individual classifiers almost always outperformed the random classifiers for all values of recall. Similarly, the individual classifiers performed very well when taking into consideration the classes that were at highest risk of containing faults.

The individual classifier results were deemed to be acceptable to use in the Ensemble Classifier and thus the evaluation of the Ensemble Classifier was performed. The F-1 results for the Ensemble Classifier are close to double that of the random classifier, a random coin toss, for both applications. This is a very good indicator that the Ensemble Classifier is able to discern the features and classify the data. Further looking at the F-10 results, they are higher than the F-1 results pointing to a higher recall, something that was missing in a number of the individual classifiers. Similarly, if we were to look at Figure 4.4, the precision of the Ensemble Classifier is able to identify a significant number of faulty classes. The precision values do not seem to go over 70%, however, it has to be remembered that the faulty dataset only account for a maximum of 35% of the complete testing dataset used, thus there exists a large data imbalance.

One concern that does arise from looking at these results is that there are other studies that have achieved a higher F-1 score, 0.75 and higher [62, 76, 99]. This is a valid concern as this system does not match the performance of previous studies in identifying faulty classes. However, it has to be noted that the dataset generation, the metrics used and the classifiers have not been tuned or manipulated in any form to help improve the performance of the framework. Since we are looking to have a versatile framework for general usage, the classifiers should not be tuned for any particular use case as this would become over-fitting. Tuning for a particular use case can also make the framework worse for other use cases.

Based on the discussion, we can confirm that it is completely viable to use an automated tool for fault prediction. The classifiers were fed information that was extracted and categorized by an automated system solely using source code history. These classifiers were then used in an ensemble learner to complete a final prediction. The results of the Ensemble Classifier were very reasonable and better than random guessing when analysis was completed using Precision-Recall curves and *Precision* @ K curves.

RQ3 Is there a correlation between performance of a fault predictor model and the resulting APFD?

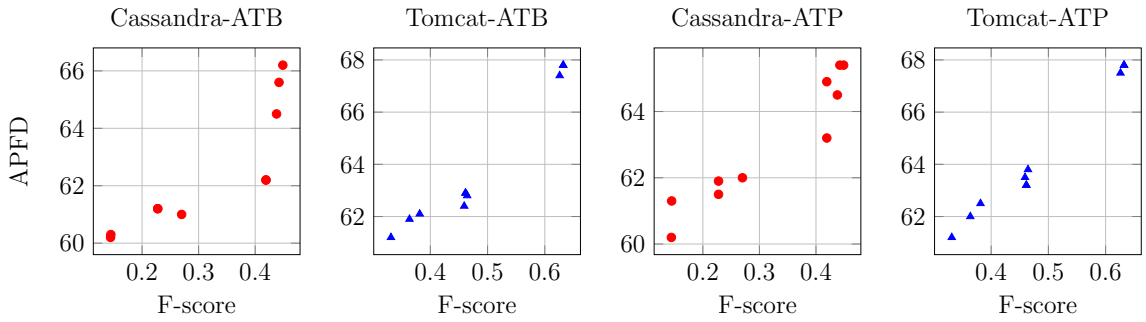

Figure 4.5: APFD of ATB and ATP in terms of the classifier F-score. Each point is associated to one of the classifiers.

So far it has been established that the supervised tie breaking schema outperforms AT in generating prioritized test suites. It has also been established that the Ensemble Classifier is able to predict fault prone classes without any human intervention or tuning. However, it still needs to be established whether there exists a correlation between the tie breaking performance and the fault prediction performance.

To evaluate this correlation, the two tie breaking schemas, ATB and ATP, were re-run using the fault-proneness returned by each individual classifier and the Ensemble Classifier. The resulting APFD values were plotted against the classifier's F-score value in a scatter plot. The resulting graphs can be seen in Figure 4.5. The figures to the left are from running ATB and the figures on the right are from running ATP.

Just from a quick glance at the results in Figure 4.5, a high correlation between F-score and APFD values can be noted. For both tie breaking schemes, the better the F-score achieved by the fault predictor, the better the resulting APFD in general. A point to note is that the Ensemble Classifier achieved a high F-score when compared to the overall performance of all the individual classifiers. This shows that the resulting APFD of the two tie breaking schemas has higher significance and is not random chance.

4.5 Threats to Validity

As with any research, there are always some threats to the validity of the results. There are either design decisions (internal), or concerns with the data used that can affect the results (external). These threats are outlined in the following sub-sections.

4.5.1 Internal Threats

There are three main internal threats to the validity of the results. They are as follows:

- Identifying Tests and Coverage - The methodology used to identify the tests within the code base was a fairly rudimentary technique. Given that the code was looked at very briefly to understand the structure of the tests, it is possible that a few test cases were missed and some coverage details might have been missed. However, spot checking after running the coverage reports did not show any missing information and thus the missing information, if any, should be minimal.
- Identifying Faulty Classes - The approach taken to identify faulty classes was to identify the commits where fixes were applied. To achieve this, any commits that had a variation of the word 'fix' in its commit message were identified as a fault fixing commit. This approach can lead to either some fixes being missed or others being wrongly marked as a fix. However, a similar approach was used by [63] successfully and thus was deemed appropriate.
- Classifier Choice - The types of classifiers used can affect the final results as discussed by [50,99]. In the realization of the framework, three of the four classifiers come from the same class of classifiers, which could be of concern. However, it is important to note that each one behaved differently with the same data, thus not causing a bias in the results. Experimenting with a different set of classifiers would be a good approach when conducting more studies.
- Lack of Data Context - The metrics aggregated for each data set was gathered at the point of time right before each fix was applied. This creates a nice snapshot of the repository before a fix, thus helping to identify what a fault might look like from a metrics perspective. However, this approach does cause a loss of context as it is not possible to include historical data. The set of events that led to a fault is not captured. Some of this information is gained back by adding in details of the number of prior commits, the number of authors and the number of co-commits in prior commits, however this data could still be incomplete. If other forms of historical data that are quantifiable could be added, it could strengthen the frameworks ability to identify faulty classes and trends that lead to faults.

4.5.2 External Threats

There were two main external threats to the validity of the results and they are as follows:

- External Tools - A number of external libraries and tools were used in the framework during the execution of the experiments. These tools are very prominent in the field of study, however they always have the possibility of containing errors. As a result, there is always the risk of the results being affected by any errors that were not captured during the realization of the framework.
- Source Code Repositories - For both applications used in the experiments, their respective git repositories were used. Git has gained a lot of popularity as a source code repository in recent years. Due to some of its superior features compared to other source code repositories and its popularity, most projects have migrated to using it. This means that older projects do not have a complete history in git, which could be of some concern. However, when comparing to studies that have completed experiments on single versions of the same applications, it can be noted that the data size is still significantly larger, thus allowing us to ignore the lost data.

4.6 Lessons Learned

In this chapter, details about the implementation of the framework have been discussed. Details about the metrics gathered and the classifiers applied have been presented along with details about the methodologies of acquiring the test suite and their code coverage. The two applications, Cassandra and Tomcat, used as case studies were outlined along with test and structure details as needed to complete the evaluation.

Evaluation of two fault driven supervised tie breaking schemas showed significant improvement over AT. The two tie breaking schemas used fault proneness values from prediction models to make their decisions. The APFD values of the prioritized test suites showed improvements of up to 5% when compared to AT. The results were achieved by a framework that was able to identify tests, gather coverage information and then prioritize the test suite based on this information.

The experiments and evaluations conducted on the Fault Prediction components of the framework have proven that it is feasible to build a predictor that needs no human intervention to predict fault proneness. It performed all its prediction tasks using information stored in a source code repository without any data grooming done prior. Most of the individual classifiers are able to discern the identifying features in the dataset and ignore most of the noise that might exist within the dataset. The results achieved from the Ensemble Classifier point towards the ability of getting a more rounded result and detection

of a larger number of trends in terms of faults within classes, resulting in reasonable precision and recall values. The Ensemble Classifier does not achieve the high precision that some of the classifiers are able to achieve in the precision-recall analysis, however there is a predictable uniformity in the results. Similarly, the Ensemble Classifier does not show very high precision when analyzing *Precision* @ K results, however they stay consistent until we get to fairly large values of K. There is some concern that the results achieved in this thesis are not as good as some other studies, however, it is hard to find any studies that have conducted fault prediction on the scale completed in this thesis.

The experiments also showed a high correlation between the performance of the fault prediction models and the performance of the tie breaking schemas. For higher F-scores from the prediction models, a higher performing test suite would be generated almost all the time. Given that the Ensemble Classifier had a F-score that was comparably higher to most of the individual classifiers, the tie breaking results become even more significant.

Chapter 5

Conclusion and Future Works

In this chapter we will revisit the contributions of this thesis and examine areas for future works.

5.1 Thesis Contributions

This thesis designed and developed an end-to-end framework to address the problem of criteria ties in coverage based TCP algorithms. The framework has been designed to only need the source code repository as an input, to be able to return prioritized test suites. In addition to its two main contributions that are detailed below, the framework performed its own test case detection and information gathering for TCP. The framework was implemented and executed against two large scale applications, Tomcat and Cassandra, as case studies. The obtained results showed consistent and significant improvement over AT.

The framework made two main contributions in the fields of TCP tie breaking and Fault Prediction. The are as follows:

Tie Breaking Mechanisms - Two tie breaking mechanisms are proposed in this thesis. Both mechanisms are supervised by fault prediction values to select the next test case when ties occur. The first mechanism was a Binary approach, whereby the fault probabilities of all classes covered by a test case, was used to break the ties. The second mechanism was a Percentage approach, whereby the fault probabilities were multiplied by the coverage percentage and then summed to break the ties. Both mechanisms showed

significant improvement and consistency when compared to AT. There was an increase of up to 5% in the APFD values in both case studies.

Fault Prediction Framework - A secondary framework was introduced inside the proposed end-to-end Framework. This internal framework is a fault predictor that is able to ingest raw information from the source code repository and make predictions based on the analysis. It is able to create datasets for training and validation of multiple supervised classifiers in an automated fashion. This dataset generation is achieved by parsing the source history, generating software metrics for the complete history, detecting all historical faults and finally forming the datasets using the fault history. The set of supervised classifiers trained with the data are finally aggregated into an ensemble learner, using weighted averages of F-1 score, for the final predications. Using an ensemble learner allowed to mitigate issues with single classifier performance and need for expert knowledge for tuning. The fault prediction framework returned an F-1 score of 0.44 and 0.63 for Cassandra and Tomcat respectively. The results are low compared to a number of prior studies, however it has to be noted that the experiments were conducted on real data and no form of data grooming or classifier tuning was performed before hand.

Analysis was also completed on the relationship between fault predictor performance and the fault driven tie breaking schemas. The results showed a high correlation between the two, whereby better predictors resulted in a better prioritized test suite. Overall, the results from the case studies have proven that breaking ties in coverage-based TCP using fault predictions return test suites that consistently have high fault detection rates.

5.2 Future Works

As any research, there are always areas of improvement or areas that can be modified to improve the works. Some of the possible improvements are outlined below.

- Use fault numbers - In the current study, fault probabilities are used to supervise the tie breaking. Instead of using fault probabilities, it could be more informative to the tie breaking schemas to use potential number of faults that exist in each module.
- Natural Language Processing - The current approach to detecting faults in the code base is fairly simple. There is possibility of misclassifying a commit as faulty or not faulty based on this technique. Instead, using natural language processing techniques could potentially improve the dataset generation for fault prediction.

- Cross project TCP - When an application is in its infancy, it will not have enough information to be able to complete defect prediction. Using a framework that has been trained on one project to predict faults on another for TCP could be an interesting avenue of research.
- Cost-effectiveness Analysis - More recent studies [58,98], investigate the cost-effectiveness of performing TCP, where cost is the execution time. The premise being a technique that takes longer to prioritize compared to the time saved from execution is not cost-effective. Performing the same analysis on the proposed tie breaking technique after implementing in a performant language would be prudent.

References

[1] ASF Git Repos - cassandra.git/summary.

[2] GitHub - apache/tomcat: Mirror of Apache Tomcat.

[3] MariaDB.org - Continuity and open collaboration.

[4] Testing at the speed and scale of Google. http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.

[5] Golnoush Abaei and Ali Selamat. A survey on software fault detection based on different prediction approaches. *Vietnam Journal of Computer Science*, 1(2):79–95, 2014.

[6] Golnoush Abaei, Ali Selamat, and Hamido Fujita. An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction. *Knowledge-Based Systems*, 74:28–39, 2015.

[7] Md Junaid Arafeen and Hyunsook Do. Test case prioritization using requirements-based clustering. In *Proceedings on IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 312–321, 2013.

[8] Hauke Baller, Sascha Lity, Malte Lochau, and Ina Schaefer. Multi-objective test suite optimization for incremental product family testing. In *Proceedings of IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST)*, pages 303–312, 2014.

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

[10] S Chatterjee, S Nigam, and A Roy. Software fault prediction using neuro-fuzzy network and evolutionary learning approach. *Neural Computing and Applications*, 28(1):1221–1231, 2017.

[11] Qian Chen and Robert A Bridges. Automated behavioral analysis of malware a case study of wannacry ransomware. *arXiv preprint arXiv:1709.08753*, 2017.

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

[13] Brad Clark and Dave Zubrow. How good is the software: a review of defect prediction techniques. In *Software Engineering Symposium, Carreige Mellon University*, 2001.

[14] Rodrigo A Coelho, Fabrício dos RN Guimarães, and Ahmed AA Esmin. Applying swarm ensemble clustering technique for fault prediction using software metrics. In *Proceedings of International Conference on Machine Learning and Applications (ICMLA)*, pages 356–361, 2014.

[15] Karel Dejaeger, Thomas Verbraken, and Bart Baesens. Toward comprehensible software fault prediction models using bayesian network classifiers. *IEEE Transactions on Software Engineering*, 39(2):237–257, 2013.

[16] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system. *Software Testing, Verification and Reliability*, 25(4):371–396, 2015.

[17] Nima Dini, Allison Sullivan, Milos Gligoric, and Gregg Rothermel. The effect of test suite type on regression test selection. In *Proceeding of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 47–58, 2016.

[18] Hyunsook Do and Gregg Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, 2006.

[19] Sepehr Eghbali and Ladan Tahvildari. Test case prioritization using lexicographical ordering. *IEEE Transactions on Software Engineering*, 42(12):1178–1195, 2016.

[20] Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering*, 28(2):159–182, 2002.

[21] Amir Elmishali, Roni Stern, and Meir Kalech. Data-augmented software diagnosis. In *AAAI*, pages 4003–4009, 2016.

[22] Wenhao Fu, Huiqun Yu, Guisheng Fan, Xiang Ji, and Xin Pei. A regression test case prioritization algorithm based on program changes and method invocation relationship. In *Proceedings on Asia-Pacific Software Engineering Conference (APSEC)*, pages 169–178, 2017.

[23] Wenhao Fu, Huiqun Yu, Guisheng Fan, Xiang Ji, and Xin Pei. A regression test case prioritization algorithm based on program changes and method invocation relationship. In *Proceeding of Asia-Pacific Software Engineering Conference (APSEC)*, pages 169–178, 2017.

[24] Jesus M Gonzalez-Barahona, Gregorio Robles, and Daniel Izquierdo-Cortazar. The metricsgrimoire database collection. In *Proceedings of IEEE/ACM Working Conference on Mining Software Repositories (MSR)*, pages 478–481, 2015.

[25] Loreto Gonzalez-Hernandez, Birgitta Lindström, Jeff Offutt, Sten F Andler, Pasqualina Potena, and Markus Bohlin. Using mutant stubbornness to create minimal and prioritized test sets. In *Proceedings on IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 446–457, 2018.

[26] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on software engineering*, 26(7):653–661, 2000.

[27] David Gray, David Bowes, Neil Davey, Yi Sun, and Bruce Christianson. The misuse of the nasa metrics data program data sets for automated software defect prediction. In *IET Conference Proceedings*, 2011.

[28] David Philip Harry Gray. Software defect prediction using static code metrics: formulating a methodology. 2013.

[29] Tihana Galinac Grbac and Darko Huljenić. On the probability distribution of faults in complex software systems. *Information and Software Technology*, 58:250–258, 2015.

[30] Stein Grimstad, Magne Jørgensen, and Kjetil Moløkken-Østvold. Software effort estimation terminology: The tower of babel. *Information and Software Technology*, 48(4):302–310, 2006.

[31] Maurice H Halstead et al. *Elements of Software Science (Operating and programming systems series)*. 1977.

[32] Dan Hao, Lingming Zhang, Lu Zhang, Gregg Rothermel, and Hong Mei. A unified test case prioritization approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):10, 2014.

[33] Dan Hao, Lu Zhang, Lei Zang, Yanbo Wang, Xingxia Wu, and Tao Xie. To be optimal or not in test-case prioritization. *IEEE Transactions on Software Engineering*, 42(5):490–505, 2016.

[34] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[35] Zhi-Wei He and Cheng-Gang Bai. Gui test case prioritization by state-coverage criterion. In *Proceedings of the 10th International Workshop on Automation of Software Test*, pages 18–22. IEEE Press, 2015.

[36] David Hovemeyer and William Pugh. Finding bugs is easy. *Acm sigplan notices*, 39(12):92–106, 2004.

[37] William E Howden. Functional program testing. *IEEE Transactions on Software Engineering*, (2):162–169, 1980.

[38] 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 *Proceedings of International Conference on Convergence Information Technology*, pages 1541–1546, 2007.

[39] Shomona Jacob and Geetha Raju. Software defect prediction in large space systems through hybrid feature selection and classification. *Int. Arab J. Inf. Technol.*, 14(2):208–214.

[40] Dennis Jeffrey and Neelam Gupta. Test case prioritization using relevant slices. In *Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International*, volume 1, pages 411–420. IEEE, 2006.

[41] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, and TH Tse. Adaptive random test case prioritization. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pages 233–244, 2009.

[42] James A Jones and Mary Jean Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, 2003.

[43] Magne Jørgensen and Kjetil Moløkken-Østvold. How large are software cost overruns? a review of the 1994 chaos report. *Information and Software Technology*, 48(4):297–301, 2006.

[44] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665, 2014.

[45] Kamaldeep Kaur and Parmeet Kaur. Evaluation of sampling techniques in software fault prediction using metrics and code smells. In *Proceedings of International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 1377–1387, 2017.

[46] Kazuya Kawata, Sousuke Amasaki, and Tomoyuki Yokogawa. Improving relevancy filter methods for cross-project defect prediction. In *Proceedings of International Conference on Applied Computing and Information Technology/2nd International Conference on Computational Science and Intelligence (ACIT-CSI)*, pages 2–7, 2015.

[47] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang NA Jawawi, and Rooster Tumeng. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 2017.

[48] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[49] Gábor Kovács, Gábor Árpád Németh, Mahadevan Subramaniam, and Zoltán Pap. Optimal string edit distance based test suite reduction for sdl specifications. In *International SDL Forum*, pages 82–97, 2009.

[50] Lov Kumar, Santanu Rath, and Ashish Sureka. Using source code metrics and ensemble methods for fault proneness prediction. *arXiv preprint arXiv:1704.04383*, 2017.

[51] Mohsen Laali, Huai Liu, Margaret Hamilton, Maria Spichkova, and Heinz W Schmidt. Test case prioritization using online fault detection information. In *Proceedings of Ada-Europe International Conference on Reliable Software Technologies*, pages 78–93, 2016.

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

[53] Yves Ledru, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. Prioritizing test cases with string distances. *Automated Software Engineering*, 19(1):65–95, 2012.

[54] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[55] Hareton KN Leung and Lee White. A study of integration testing and software regression at the integration level. In *Procedings of International Conference on Software Maintenance (ICSM)*, pages 290–301, 1990.

[56] Sihan Li, Naiwen Bian, Zhenyu Chen, Dongjiang You, and Yuchen He. A simulation study on some search algorithms for regression test case prioritization. In *Proceedings of International Conference on Quality Software*, pages 72–81, 2010.

[57] Zheng Li, Mark Harman, and Robert M Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on software engineering*, 33(4), 2007.

[58] J. Liang, S. Elbaum, and G. Rothermel. Redefining prioritization: Continuous prioritization for continuous integration. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 688–698, May 2018.

[59] Yiling Lou, Dan Hao, and Lu Zhang. Mutation-based test-case prioritization in software evolution. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 46–57. IEEE, 2015.

[60] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. How does regression test prioritization perform in real-world software evolution? In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 535–546. IEEE, 2016.

[61] R Uma Maheswari and D Jeya Mala. Combined genetic and simulated annealing approach for test case prioritization. *Indian Journal of Science and Technology*, 8(35).

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

[63] Ruchika Malhotra, Laavanye Bahl, Sushant Sehgal, and Pragati Priya. Empirical comparison of machine learning algorithms for bug prediction in open source software. In *Proceedings of International Conference on Big Data Analytics and Computational Intelligence (ICBDAC)*, pages 40–45, 2017.

[64] Ruchika Malhotra and Rajeev Raje. An empirical comparison of machine learning techniques for software defect prediction. In *Proceedings of International Conference on Bioinspired Information and Communications Technologies*, pages 320–327, 2014.

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

[66] Hong Mei, Dan Hao, Lingming Zhang, Lu Zhang, Ji Zhou, and Gregg Rothermel. A static approach to prioritizing junit test cases. *IEEE Transactions on Software Engineering*, 38(6):1258–1275, 2012.

[67] Lijun Mei, Yan Cai, Changjiang Jia, Bo Jiang, Wing Kwong Chan, Zhenyu Zhang, and TH Tse. A subsumption hierarchy of test case prioritization for composite services. *IEEE Transactions on Services Computing*, 8(5):658–673, 2015.

[68] T. Menzies, R. Krishna, and D. Pryor. The promise repository of empirical software engineering data.., 2015.

[69] 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.

[70] Breno Miranda, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino. Fast approaches to scalable similarity-based test case prioritization. In *Proceedings of International Conference on Software Engineering*, pages 222–232, 2018.

[71] Siavash Mirarab, Soroush Akhlaghi, and Ladan Tahvildari. Size-constrained regression test case selection using multicriteria optimization. *IEEE transactions on Software Engineering*, 38(4):936–956, 2012.

[72] Audris Mockus and Lawrence G Votta. Identifying reasons for software changes using historic databases. In *ICSM*, pages 120–130, 2000.

[73] K-H Moller and Daniel J Paulish. An empirical investigation of software fault distribution. In *Proceedings of International Software Metrics Symposium*, pages 82–90, 1993.

[74] Fachrul Pralienka Bani Muhamad, Daniel Oranova Siahaan, and Chastine Fatichah. Software fault prediction using filtering feature selection in cluster-based classification. *IPTEK Journal of Proceedings Series*, (1):59–64.

[75] Reetika Nagar, Arvind Kumar, Gaurav Pratap Singh, and Sachin Kumar. Test case selection and prioritization using cuckoos search algorithm. In *Proceedings of International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*, pages 283–288, 2015.

[76] Jaechang Nam and Sunghun Kim. Clami: Defect prediction on unlabeled datasets (t). In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 452–463, 2015.

[77] Akbar Siami Namin and Sahitya Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 342–352, 2011.

[78] Tanzeem Bin Noor and Hadi Hemmati. A similarity-based approach for test case prioritization using historical failure data. In *Proceedings of IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 58–68, 2015.

[79] Tanzeem Bin Noor and Hadi Hemmati. Studying test case failure prediction for test case prioritization. In *Proceedings of International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE, pages 2–11, 2017.

[80] João Felipe Silva Ouriques, Emanuela Gadelha Cartaxo, and Patrícia Duarte Lima Machado. Revealing influence of model structure and test case profile on the prioritization of test cases in the context of model-based testing. *Journal of Software Engineering Research and Development*, 3(1):1, 2015.

[81] Deepak Panwar, Pradeep Tomar, Harshvardhan Harsh, and Mohammad Husnain Siddique. Improved meta-heuristic technique for test case prioritization. In *Soft Computing: Theories and Applications*, pages 647–664. 2018.

[82] David Paterson, Gregory M Kapfhammer, Gordon Fraser, and Phil McMinn. Using controlled numbers of real faults and mutants to empirically evaluate coverage-based test case prioritization. In *Proceedings of the International Workshop on Automation of Software Test (AST)*, 2018.

[83] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[84] Jean Petrić, David Bowes, Tracy Hall, Bruce Christianson, and Nathan Baddoo. Building an ensemble for software defect prediction based on diversity selection. In *Proceedings of International Symposium on Empirical Software Engineering and Measurement*, page 46, 2016.

[85] Santosh S Rathore and Sandeep Kumar. A study on software fault prediction techniques. *Artificial Intelligence Review*, pages 1–73, 2017.

[86] Santosh Singh Rathore and Sandeep Kumar. Towards an ensemble based system for predicting the number of software faults. *Expert Systems with Applications*, 82:357–382, 2017.

[87] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Proceedings of IEEE International Conference on Software Maintenance (ICSM)*, pages 179–188, 1999.

[88] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948, 2001.

[89] Shounak Roychowdhury. Ensemble of feature selectors for software fault localization. In *Proceedings of IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1351–1356, 2012.

[90] S. Sampath, R. C. Bryce, G. Viswanath, V. Kandimalla, and A. G. Koru. Prioritizing user-session-based test cases for web applications testing. In *Proceedings on International Conference on Software Testing, Verification, and Validation*, pages 141–150, 2008.

[91] Sreedevi Sampath, Renee Bryce, and Atif M Memon. A uniform representation of hybrid criteria for regression testing. *IEEE Transactions on Software Engineering*, 39(10):1326–1344, 2013.

[92] Deepak Sharma and Pravin Chandra. Software fault prediction using machine-learning techniques. In *Smart Computing and Informatics*, pages 541–549. 2018.

[93] Pradeep Singh, Nikhil R Pal, Shrish Verma, and Om Prakash Vyas. Fuzzy rule-based approach for software fault prediction. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 47(5):826–837, 2017.

[94] Carlos Soares, Pavel B Brazdil, and Petr Kuba. A meta-learning method to select the kernel width in support vector regression. *Machine learning*, 54(3):195–209, 2004.

[95] Qinbao Song, Zihan Jia, Martin Shepperd, Shi Ying, and Jin Liu. A general software defect-proneness prediction framework. *IEEE Transactions on Software Engineering*, 37(3):356–370, 2011.

[96] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 12–22, 2017.

[97] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 2, pages 99–108, 2015.

[98] Y. Tsutano, S. Bachala, W. Srisa-An, G. Rothermel, and J. Dinh. An efficient, robust, and scalable approach for analyzing interacting android apps. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 324–334, May 2017.

[99] 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.

[100] Kristen R Walcott, Mary Lou Soffa, Gregory M Kapfhammer, and Robert S Roos. Timeaware test suite prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 1–12, 2006.

[101] Hua Wang, Cuiqin Ma, and Lijuan Zhou. A brief review of machine learning and its application. In *Proceedings of International Conference on Information Engineering and Computer Science (ICIECS)*, pages 1–4, 2009.

[102] Song Wang, Jaechang Nam, and Lin Tan. Qtep: quality-aware test case prioritization. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 523–534. ACM, 2017.

[103] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[104] Chubato Wondaferaw Yohannese and Tianrui Li. A combined-learning based framework for improved software fault prediction. *International Journal Of Computational Intelligence Systems*, 10(1).

[105] Yuen Tak Yu and Man Fai Lau. Fault-based test suite prioritization for specification-based testing. *Information and Software Technology*, 54(2):179–202, 2012.

[106] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of the International Conference on Software Engineering*, pages 192–201, 2013.

[107] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997.

APPENDICES

Appendix A

Metrics Description

Following is the set of metrics, as they appear in the database, and a description of the data they hold grouped by class of metrics. All metrics descriptions, other than the History set, have been retrieved from the scitools website.

Table A.1: Count Metrics

Metric Name	**Metric Description**
altcountlineblank	Number of blank lines, including inactive regions
altcountlinecode	Number of lines of code, including inactive regions
altcountlinecomment	Number of lines of comments, including inactive regions
countdeclfile	Number of files
countdeclfilecode	Number of code files
countdeclfileheader	Number of header files
countdeclfunction	Number of functions
countdeclinstancevariableinternal	Number of internal variables
countdeclinstancevariableprotectedinternal	Number of protected internal variables
countdecllibunit	Number of library units
countdeclmethodfriend	Number of local friend methods
countdeclmethodinternal	Number of local internal methods
countdeclmethodprotectedinternal	Number of protected internal methods

countinput	Number of calling subprograms plus global variables read.
countline	Number of all lines
countlineblank	Number of blank lines
countlinecode	Number of lines containing source code
countlinecodedecl	Number of lines containing declarative source code
countlinecodeexe	Number of lines containing executable source code
countlinecomment	Number of lines containing comments
countlineinactive	Number of inactive lines
countlinepreprocessor	Number of preprocessor lines
countsemicolon	Number of semicolons
countstmt	Number of statements
countstmtdecl	Number of declarative statements
countstmtempty	Number of empty statements
countstmtexe	Number of executable statements

Table A.2: Chidamber and Kemerer Metrics

Metric Name	**Metric Description**
countclassbase	Number of immediate base classes
countclasscoupled	Number of other classes coupled
countclassderived	Number of immediate subclasses
countdeclclass	Number of classes
countdeclclassmethod	Number of class methods
countdeclclassvariable	Number of class variables
countdeclinstancemethod	Number of instance methods
countdeclinstancevariable	Number of instance variables
countdeclinstancevariableprivate	Number of private instance variables
countdeclinstancevariableprotected	Number of protected instance variables
countdeclinstancevariablepublic	Number of public instance variables
countdeclmethod	Number of local methods
countdeclmethodall	Number of local methods, including inherited
countdeclmethodconst	Number of local const methods

countdeclmethoddefault	Number of local default methods
countdeclmethodprivate	Number of local private methods
countdeclmethodprotected	Number of local protected methods
countdeclmethodpublic	Number of local public methods
countdeclmethodstrictprivate	Number of local strict private methods
countdeclmethodstrictpublished	Number of local strict published methods
countdeclmodule	Number of modules
countdeclprogunit	Number of non-nested modules, block data units and subprograms
countdeclsubprogram	Number of subprograms
countoutput	Number of called subprograms plus global variables set
countpackagecoupled	Number of other packages coupled to
maxinheritancetree	Maximum depth of class in inheritance tree
percentlackofcohesion	100% minus the average cohesion for package entities
loc	Number of lines of code
sloc	Number of source lines of code

Table A.3: McCabe Metrics

Metric Name	**Metric Description**
countpath	Number of possible paths, not counting abnormal gotos or exits
cyclomatic	Cyclomatic complexity
cyclomaticmodified	Modified cyclomatic complexity
cyclomaticstrict	Strict cyclomatic complexity
essential	Essential Complexity
essentialstrictmodified	Strict modified essential complexity
maxcyclomatic	Maximum cyclomatic complexity of all nested methods or functions
maxcyclomaticmodified	Maximum modified cyclomatic complexity of all nested methods or functions
maxcyclomaticstrict	Maximum strict cyclomatic complexity of all nested methods or functions

maxessential	Maximum essential cyclomatic complexity of all nested methods or functions
maxessentialstrictmodified	Maximum strict modified essential cyclomatic complexity of all nested methods or functions
maxnesting	Maximum nesting level of control constructs
sumcyclomatic	Sum of cyclomatic complexity of all nested functions or methods
sumcyclomaticmodified	Sum of modified cyclomatic complexity of all nested functions or methods
sumcyclomaticstrict	Sum of strict cyclomatic complexity of all nested functions or methods
sumessential	Sum of essential cyclomatic complexity of all nested functions or methods
sumessentialstrictmodified	Sum of strict modified essential cyclomatic complexity of all nested functions or methods
knots	Measure of overlapping jumps
maxessentialknots	Maximum Knots after structured programming constructs have been removed
MI	Maintainability Index
minessentialknots	Minimum Knots after structured programming constructs have been removed

Table A.4: Halstead Metrics

Metric Name	**Metric Description**
D	Difficulty of program
E	Effort to maintain
n	Program vocabulary
N_	Program length
N_1	Total number of operators
N_2	Total number of operands
n1	Number of distinct operators
n2	Number of distinct operands

V	Program Volume

Table A.5: History Metrics

Metric Name	**Metric Description**
Authors_Since_Fix	Number of authors editing code since last fix applied
Avg_Co_Commits	Average number of classes co-commited since start of time
Avg_Co_Commits_Since_Fix	Average number of classes co-commited since last fix applied
Co_Commits_Since_Fix	Total number of co-commits since last fix applied
Commits_Since_Fix	Total number of times class has been modified since last fixed
Total_Authors	Total number of authors modified class since start of time
Total_Co_Commits	Total number of classes co-committed since start of time
Total_Commits	Total number of time class has been modified since start of time