CAPITAL UNIVERSITY OF SCIENCE AND TECHNOLOGY, ISLAMABAD

Evaluating the Effectiveness of Decomposed Halstead Metric Suite in Software Fault Prediction

by

Muhammad Bilal Khan

A dissertation submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Computing

Department of Computer Science

2025

Evaluating the Effectiveness of Decomposed Halstead Metric Suite in Software Fault Prediction

By

Muhammad Bilal Khan

(DCS171003)

DEPARTMENT OF COMPUTER SCIENCE

CAPITAL UNIVERSITY OF SCIENCE AND TECHNOLOGY

ISLAMABAD

2025

To my parents, siblings, wife, and five angels, Hiba, Fasih, Shaheer, Anas and Aiza.

CAPITAL UNIVERSITY OF SCIENCE & TECHNOLOGY
ISLAMABAD
Expressway, Kahuta Road, Zone-V, Islamabad
Phone: +92-51-111-555-666 Fax: +92-51-4486705
Email: info@cust.edu.pk Website: https://www.cust.edu.pk

CERTIFICATE OF APPROVAL

This is to certify that the research work presented in the dissertation, entitled "**Evaluating the Effectiveness of Decomposed Halstead Metric Suite in Software Fault Prediction**" was conducted under the supervision of **Dr. Aamer Nadeem**. No part of this dissertation has been submitted anywhere else for any other degree. This dissertation is submitted to the **Department of Computer Science, Capital University of Science and Technology** in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the field of **Computer Science.** The open defence of the dissertation was conducted on **December 27, 2024**.

Student Name:	**Muhammad Bilal Khan** (DCS171003)	[illegible]

The Examination Committee unanimously agrees to award PhD degree in the mentioned field.

Examination Committee:

(a)	External Examiner 1:	Dr. Onaiza Maqbool Professor QAU, Islamabad	[illegible]
(b)	External Examiner 2:	Dr. Tamim Ahmed Khan Professor Bahria University, Islamabad	[illegible]
(c)	Internal Examiner:	Dr. Nadeem Anjum Professor CUST, Islamabad	[illegible]

Supervisor Name:	Dr. Aamer Nadeem Professor CUST, Islamabad	[illegible]
Name of HoD :	Dr. Abdul Basit Siddiqui Associate Professor CUST, Islamabad	[illegible]
Name of Dean:	Dr. Muhammad Abdul Qadir Professor CUST, Islamabad	[illegible]

<u>AUTHOR'S DECLARATION</u>

I, **Muhammad Bilal Khan (Registration No. DCS171003)**, hereby state that my dissertation titled, '**Evaluating the Effectiveness of Decomposed Halstead Metric Suite in Software Fault Prediction**' is my own work and has not been submitted previously by me for taking any degree from Capital University of Science and Technology, Islamabad or anywhere else in the country/ world.

At any time, if my statement is found to be incorrect even after my graduation, the University has the right to withdraw my PhD Degree.

[illegible]

(**Muhammad Bilal Khan**)

Dated: 27 December, 2024

Registration No: DCS171003

PLAGIARISM UNDERTAKING

I solemnly declare that research work presented in the dissertation titled "**Evaluating the Effectiveness of Decomposed Halstead Metric Suite in Software Fault Prediction**" is solely my research work with no significant contribution from any other person. Small contribution/ help wherever taken has been duly acknowledged and that complete dissertation has been written by me.

I understand the zero-tolerance policy of the HEC and Capital University of Science and Technology towards plagiarism. Therefore, I as an author of the above titled dissertation declare that no portion of my dissertation has been plagiarized and any material used as reference is properly referred/ cited.

I undertake that if I am found guilty of any formal plagiarism in the above titled dissertation even after award of PhD Degree, the University reserves the right to withdraw/ revoke my PhD degree and that HEC and the University have the right to publish my name on the HEC/ University Website on which names of students are placed who submitted plagiarized dissertation.

[illegible]

(Muhammad Bilal Khan)

Dated: 27 December, 2024

Registration No: DCS171003

List of Publications

It is certified that following publication has been made out of the research work that has been carried out for this dissertation:-

1. B. Khan and A. Nadeem, “Evaluating the effectiveness of decomposed halstead metrics in software fault prediction,” PeerJ Computer Science, vol. 9, p. e1647, 2023.

Muhammad Bilal Khan
(DCS171003)

Acknowledgement

All praise is due to Allah Almighty, who has all knowledge and has the power to grant from his knowledge and peace, mercy and blessing upon his last and final messenger and upon good-doers.

I would never have been able to finish my work without the support from my respected parents and after that my younger sister for her moral support and encouragement.

One of the most notable of Allah's blessings upon me was in the form of my supervisor. I would like to thank my supervisors, Prof. Dr. Aamer Nadeem for his guidance. I am also grateful to Dr. Muddassar Azam Sindhu, Assistant Professor Quaid-i-Azam University, whose reviews help me a lot to improve my work. I feel very lucky to be part of CSD research group members whose discussion and constructive criticism maintained an environment that was conducive for research. Moreover, without the recreational activities of our CSD research group, I may have gone insane over the last few years. I also thank the faculty members of Capital University of Science and Technology who gave me the resources and healthy education environment.

Very special thanks to my siblings for their motivation and encouragement. I would like to thank my beloved *Wife*. She was always there cheering me up and stood by me through the good times and bad.

I would like to express my gratitude to my friend and colleague Dr.Muhamad Rizwan for his technical guidance. Apart from the mentioned above, many people helped me reach this point. May Allah grant all of them peace and bless them with prosperity.

Abstract

Occurrence of faults in software systems is an inescapable problem. Testing helps in identifying those faults, however, exhaustive testing is required to identify all the residual faults, which is infeasible for any nontrivial system. Software fault prediction (SFP) refines the process of testing while streamlining the effort to review/test the code. There are several approaches for SFP, Machine learning (ML) is the most dominating one. ML based SFP uses some metrics, which could be file level, class level, method level, or even line level. More granulated metric is expected to have micro coverage of the code. Halstead metric suite provides line level coverage of code. It has been used in various domains like, fault prediction, quality assessment, similarity approximation for last three decades. Keeping in view its reported effectiveness, this dissertation aims to enhance its predictive ability through decomposition. Decomposition refers to the splitting of base Halstead metrics into multiple metrics. We decomposed Halstead base metrics (Operators/Operands) at multiple levels to evaluate improvement in fault prediction at each level. In order to test the effectiveness of our multilevel decomposed Halstead based metrics empirically, five publicly accessible datasets with instances classified as fault prone and not fault prone were used in each experiment. Logistic regression, Naive Bayes, Decision Trees, Multilayer Perceptron, Random Forests, and Support Vector Machines were all used in the machine learning modeling. These models' performance was evaluated using Area Under the Curve (AUC), Accuracy, and F-measure measures. We have performed four experiments with decomposed Halstead base metric combine with Halstead derived, McCabe and LoC metrics suites that are frequently used in analysis related to software fault prediction with Halstead. In the first experiment, we employed the conventional Halstead base metrics. In the second experiment, decomposed Halstead operators at level 1 were combined with conventional operands. The third experiment utilized both decomposed Halstead operators and operands at level 1, while the fourth experiment integrated decomposed Halstead operators at level 2 and operands at level 1, in combination with derived Halstead, McCabe, and Lines of Code (LoC) metrics as predictors. The results of these experiments demonstrate the effectiveness of decomposed Halstead operators and operands at level 1, however, decrease in the results are observed in decomposed operators at level 2.

Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial intelligence
AM	Association rule Mining
ANN	Artificial Neural Networks
AUC	Area Under Curve
DC	Data Collection
DHM	Decomposed Halstead base Metrics
DIT	Depth of Inheritance Tree
DM	Deployment and Monitoring
DT	Decision Tree
EO	Experts' Opinion
FE	Feature Extraction
FNR	False Negative Rate
IV	Independent Variables
k-NN	K-Nearest Neighbor
Lb	Labeling
LOC	Lines of code
LR	Logistic Regression
LSSVM	Least Squares Support Vector Machines
ME	Model Evaluation
ML	Machine Learning
MST	Model Selection and Training
NASA	National Aeronautics and Space Administration
NB	Naïve Bayes
NECM	Normalized Expected Cost of Misclassification

NN	Neural Network
PFA	Probability of False Alarm
RF	Random Forest
RQ	Research Question
SDLC	Software Development Life Cycle
SFP	Software Fault Prediction
SLMBC	Spiral life cycle model-based Bayesian classification
SLR	Systematic literature review
SM	Statistical Modeling
SVM	Support Vector Machine
TDP	Training Data Preparation

Chapter 1

Introduction

A crucial part of the Software Development Life Cycle (SDLC) is software testing, which is the methodical assessment of software to make sure it satisfies requirements and operates as intended [1–3]. Its ability to find flaws, faults, or problems early in the development process reduces the need for expensive modifications and guarantees that end customers will receive a dependable, high-quality product [4]. This is why software testing is crucial to the SDLC. Functional testing, performance testing, security testing, and other approaches and procedures are all included in software testing [5–12]. Utilizing predictive analytics approaches [13], Software Fault Prediction (SFP) helps software testers identify potential flaws or problems in software systems before they arise [3]. According to the study conducted by Timothy *et al.*, in a software system faults are found in only 42% of the software modules[14] [15]. Likewise, another study reported that about 70% of the faults are found by testing only 6% of the software modules[16]. Similar results are also reported by multiple studies conducted by Weyuker *et al.* [17, 18].

SFP aids in resource allocation, testing effort prioritization, and focusing attention on software components most likely to contain flaws by examining past data and spotting trends. According to [19, 20], testing and quality assurance account for roughly 35% of the overall development cost. SFP integration improves efficacy and efficiency in the testing process, which lowers development costs and improves software quality [19]. Additionally, it makes proactive risk management possible, allowing development teams to address possible problems early in the development lifecycle and provide users with more dependable and robust software solutions [12].

1.1 Software Fault Prediction

A crucial step in software engineering, Software Fault Prediction (SFP) identifies modules that are prone to errors or calculates the total amount of faults in a software system [21, 22]. Timely identification of malfunctioning modules or the forecasting of malfunctions within modules is advantageous, especially when it comes to vital and tactical software systems. It plays a crucial role in improving the system's overall quality and reliability in addition to helping to lower testing expenses [22, 23]. SFP's ability to direct testing efforts toward the identification and mitigation of fault-prone modules is one of its main features. SFP maximizes the efficacy of testing procedures and facilitates optimal resource allocation by prioritizing testing activities based on expected fault probabilities [24, 25]. Additionally, forecasting the number of errors in a module offers insightful information about how well tests are done, assisting teams in determining when sufficient testing coverage has been attained. Several strategies have been created and used in the field of SFP to address the problem of software fault prediction. These strategies cover a variety of techniques.

1.1.1 Machine Learning (ML)

As a branch of artificial intelligence, machine learning gives machines the ability to learn from data and carry out activities that need human knowledge. It includes a range of methods for extracting patterns and insights from data to support automated decision-making [26]. Supervised learning [27], which involves learning from labeled data, unsupervised learning [28], which involves learning from unlabeled data, and reinforcement learning [29], which involves learning from feedback, are the three primary forms. In supervised learning, predictions are created by matching input data with labels. Unsupervised learning is helpful for jobs like anomaly detection because it may identify patterns in data such as associations or outliers. To maximize rewards, reinforcement learning entails learning from actions and feedback. This technique is helpful in control systems and decision-making. In software development, machine learning is used for tasks like defect prediction. Data such as version control history and bug reports are analyzed using algorithms.

1.1.2 Statistical Modeling (SM)

Predicting software faults is based on statistical modeling, which creates models by examining past data [30]. By estimating the probability of flaws, statistical methods increase the precision of fault prediction. These models improve system resilience and efficiency by facilitating early defect detection, pattern recognition, and resource allocation optimization in software development [31–33].

1.1.3 Expert Opinion (EO)

Expert judgment is essential in software fault prediction to pinpoint problematic software modules or code segments [34]. Several methods have been devised to integrate these viewpoints, augmenting the precision of fault prediction approaches [34]. These methods improve software system reliability by enabling proactive issue detection through the combination of human expertise and computational algorithms.

1.1.4 Similarity Based (SB)

By employing source code similarity to forecast possible mistakes, software fault prediction through similarity analysis increases testing accuracy in complex software systems [35]. Techniques include using probabilistic modeling to assess the likelihood of faults in individual components and calculating similarity metrics based on syntax and semantics. Development teams can improve system dependability and optimize resource allocation across the software development life cycle by using these strategies to proactively resolve probable faults.

1.1.5 Association Rule Mining (AM)

Software defect prediction relies on association rule mining, a fundamental data analysis technique that finds important patterns in large datasets by defining rules based on program parameters like lines of code and object coupling [36]. This procedure makes it easier to identify rules, improves prediction accuracy, and makes it possible to manage intricate interactions in dynamic software systems [37]. Software development teams can learn more about the causes of fault occurrences by assessing the importance of rules that are found. Metrics are also used to evaluate rule quality, which enhances the quality and dependability of software systems.

Machine learning (ML) is a popular choice for software fault prediction (SFP) [38]. Metrics and fault data, which might be continuous or discrete (binary or ordinal), are essential to SFP. Regression-based techniques like Linear Regression handle continuous defect data [15, 39], while classification-based ML methods like Decision Trees and Neural Networks handle discrete defect data. In SFP, the emphasis is frequently on classifying modules as faulty or not, allowing for focused interventions to raise the caliber of software. The goal of machine learning algorithms customized for defect data is to improve forecast accuracy and support software development workflow decision-making.

1.2 Machine Learning

In predictive analytics on software defects, machine learning plays a crucial role in software fault prediction by helping to identify faults before they happen [40]. This proactive strategy lowers development costs, lessens risks, and improves the dependability and quality of software. Usually, the procedure proceeds through some linked stages (Show in Fig 1.1):

1.2.1 Data Collection (DC)

The gathering of comprehensive and diversified datasets related to software development processes is the cornerstone of software fault prediction [40]. Code metrics, version control information (commits, branches), bug reports, testing outcomes, and developer activity are just a few of the important pieces of data that are included in these datasets. Predictive models are built upon this data as their foundation.

1.2.2 Feature Extraction (FE)

After data collection, essential characteristics are taken out of the collected datasets to capture important aspects of the software development environment [40]. These features cover a wide range of topics, such as developer experience, previous issue complaints linked to particular modules or components, code complexity metrics, and code churn (frequency of code changes). The next step in the modeling process depends critically on the extraction of discriminating characteristics.

1.2.3 Labeling (Lb)

Alongside feature extraction, the collected data is carefully labeled to indicate whether or not specific software modules or components have bugs [40]. This labeling procedure classifies cases based on previous problem reports or testing results. In later stages, supervised learning algorithms rely heavily on the precise labeling of data instances.

1.2.4 Training Data Preparation(TDP)

To aid in the training and assessment of machine learning models, the labeled data is divided into separate training and testing datasets [40]. By exposing prediction models to labeled examples, the training dataset acts as a crucible for their refinement. On the other hand, the testing dataset makes it possible to evaluate the generalization and performance of the model on untested data.

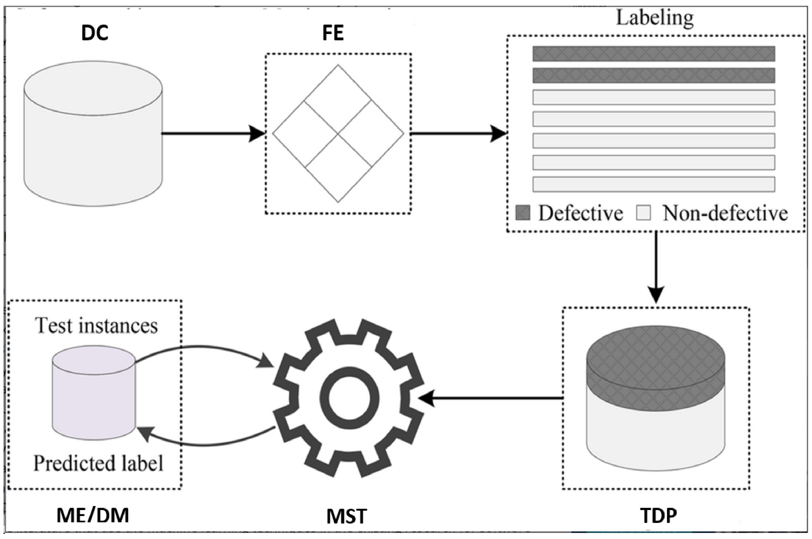

FIGURE 1.1: General procedure proceeds in ML

1.2.5 Model Selection and Training (MST)

For software fault prediction tasks, a wide range of machine learning techniques can be used, such as logistic regression, decision trees, random forests, support vector machines, and neural networks [40]. These algorithms iterate over the training dataset to identify complex correlations and patterns between input feature existence and software component defects. The choice of the best model depends on factors including computing efficiency, interpretability, and performance.

1.2.6 Model Evaluation (ME)

After training the model, a thorough assessment is conducted using the testing dataset to determine the prediction models' reliability and effectiveness [40]. Numerous assessment criteria, including receiver operating characteristic (ROC) curve analysis, accuracy, precision, recall, and F1-score, provide quantifiable information about how well machine learning models predict the future.

1.2.7 Deployment and Monitoring (DM)

After achieving an acceptable model, it is deployed in real-world software development settings to initiate ongoing code change monitoring and fault-prone module or component identification [40]. Predictive models can be adaptively refined to take into account changing software development techniques and trends through regular monitoring and retraining cycles. Organizations can proactively strengthen their software ecosystems against potential defects by utilizing machine learning techniques in software fault prediction endeavors. This can improve software quality, reduce development costs, and strengthen overall software reliability and maintainability. The deliberate incorporation of predictive analytics portends well for the further use of flexible and robust software development methodologies in modern environments.

1.3 Software Metrics

Software metrics are indispensable tools in the realm of software development, serving as the foundation for understanding, evaluating, and improving various aspects of software projects [41]. These metrics, ranging from measures of code complexity to indicators of code quality and performance, offer invaluable insights that guide developers, project managers, and stakeholders in making informed decisions [42]. By harnessing software metrics, SFP can proactively detect potential faults before they escalate into full-blown faults [43] [44] [45]. Early identification of problematic areas allows for timely intervention, reducing the likelihood of costly rework and delays in the development timeline [43]. Moreover, SFP using software metrics enables teams to prioritize testing efforts by directing resources toward the most critical and high-risk areas of the code. Through software metrics SFP enhances the effectiveness of testing processes, resulting in more robust and reliable software products [46].

Beyond defect prediction, software metrics play a pivotal role in quantifying code quality and stability. Metrics such as code complexity, code coverage, and code duplication provide objective measures of the health of the codebase, guiding efforts to refactor and optimize code for improved maintainability and reliability. Additionally, software metrics facilitate data-driven decision-making, empowering project managers [47] and stakeholders to allocate resources efficiently, manage risks effectively, and drive the project towards successful outcomes[48]. In essence, in SFP software metrics serve as the compass that navigates software development endeavors, guiding teams toward the creation of high-quality, fault-tolerant software solutions [49]. Embracing the insights gleaned from these metrics empowers teams to mitigate risks, streamline processes, and ultimately deliver software that meets the highest standards of excellence [50].

1.3.1 Product Metrics

Software product metrics are numerical measurements that are used to evaluate a software product's size, complexity, quality, performance, and maintainability, among other characteristics [51]. Software volume is measured by size measures like function points (FP) and lines of code (LOC). Complexity measures, such as Halstead and cyclomatic complexity, assess the complexity of code to forecast fault-proneness and maintainability. Defect density and reliability are examples of quality metrics that quantify the quantity of flaws and dependability of a system. While maintainability measures like code churn and coupling measure how easy it is to update the software, performance metrics evaluate reaction time and throughput. Additional measures of component adaptation and program resilience against vulnerabilities are provided by reusability and security metrics. When combined, these indicators offer thorough insights to enhance the quality, maintainability, and overall effectiveness.

1.3.2 Process Metrics

Process metrics, which include development time, and coding standard adherence, show how effective and efficient the software development process is [52]. These indicators guide attempts to improve productivity and optimize processes by providing insights into the state and effectiveness of the development process.

1.3.3 Project Metrics

Project metrics assess variables including resource use, budget adherence, and schedule compliance, and offer insight into project-specific dynamics [53]. Project metrics provide project managers with the ability to monitor progress and make well-informed strategic decisions to ensure project success by providing them with visibility into key performance indicators, such as resource allocation and milestones achieved.

To sum up, software metrics are essential instruments for assessing, tracking, and enhancing software development projects. Organizations can improve their capacity to produce high-quality software products quickly and effectively, which will lead to overall success in software development projects, by utilizing product, process, and project metrics.

1.4 Significance of Software Metrics in SFP

SFP relies heavily on the combination of machine learning and software metrics [43]. These metrics measure several aspects of the software and help find places that are prone to errors [54]. By automating the discovery of patterns in historical data, machine learning makes it possible to create prediction models that identify software components that are prone to errors. Combining machine learning techniques with software measurements to detect faults has several advantages [54]:

- *Improved prediction accuracy.*
- *Early detection of faults.*
- *Resource optimization.*
- *Continuous improvement.*

In summary, the combination of software metrics with machine learning techniques enables organizations to proactively detect and fix software defects, resulting in the development of high-caliber software products and enhancing the dependability of software systems. Employing this cooperative integration, companies set out to cultivate a proactive fault management strategy, which in turn leads to previously unheard-of levels of efficacy and durability for their software engineering initiatives.

1.5 Research Objectives

Investigating the breakdown of Halstead base metrics and their use in SFP is the main goal of this study. This study aims to fill in the gaps in existing approaches and increase the precision of prediction models in detecting software components that are prone to faults by utilizing a systematic deconstruction methodology. The following are the goals of this study:

1.5.1 Decompose Halstead Base Metrics for Enhanced Predictive Accuracy

Examine the viability and efficiency of breaking down Halstead base metrics into more manageable parts, such as operators and operands. The goal of this procedure is to produce a more comprehensive feature set that improves machine learning models' ability to predict software faults.

1.5.2 Determine Optimal Decomposition Levels

Investigate several decomposition levels to determine the ideal granularity that produces the best prediction accuracy. To guarantee practical applicability, this goal entails evaluating the trade-offs between computing efficiency and predictive performance.

1.5.3 Quantify the Impact on Fault Prediction Precision

Examine the effects of deconstructed Halstead metrics on fault prediction models' accuracy. This entails assessing how well they differentiate between components that are prone to errors and those that are not, as well as comprehending the consequences for software testing methodologies.

1.5.4 Validate Comparative Models

Use both traditional and decomposed Halstead measures to validate prediction models. To find out if decomposition significantly improves predictive performance especially in datasets with different levels of complexity a comparison analysis will be carried out.

1.5.5 Enhance Quality Assurance Practices in Software Engineering

Give practitioners of software engineering practical advice and insights. The accuracy and effectiveness of quality assurance procedures can be improved by showcasing how deconstructed metrics can be incorporated into current fault prediction methods.

1.5.6 Advance the Understanding of Metric Decomposition in SFP

Examine the theoretical foundations and real world uses of metric decomposition to add to the corpus of knowledge in software defect prediction. The goal of this study is to lay the groundwork for further research in this area. By fulfilling these goals, the study not only fills in knowledge gaps in the area but also offers useful resources and perspectives to practitioners in the industry, thereby advancing the discipline of software engineering.

1.6 Problem Statement

Feature decomposition to improve ML models predictiveness is proven except in SFP. The role of SFP is crucial in ensuring software reliability. The research community has made significant efforts to enhance SFP by selecting effective ML models and identifying optimal combinations of software metrics. However, the effectiveness of metric decomposition in SFP remains under explored in the literature. While the Halstead metric suite has been widely recognized for its contribution to software fault prediction since its inception in the 1970s, existing studies have predominantly utilized these metrics in their original form, employing operators and operands as is. The potential of decomposing operators and operands to provide more granular insights, thereby improving the performance of machine learning models, has not been adequately investigated in current research.

1.7 Proposed Idea

Software testing is an essential part of the software development life cycle [55] and has a big impact on the final product's quality and dependability [56]. In this field, software fault prediction shows up as an essential instrument for anticipating and fixing possible errors [57]. Although research is still being done, improving forecast accuracy is still the major goal [58]. The current research efforts are focused on improving the efficacy of software fault prediction using a variety of strategies, including improvements in machine learning algorithms, quality dataset optimization, and the tactical incorporation of software metrics for fault prediction [59–61]. The Halstead metric stands out as a significant contributor, demonstrating a high degree of effectiveness in previous fault prediction attempts [62–69]. However, a fascinating question emerges: is it possible to improve software defect prediction even more by decomposing the Halstead metric? Although there is theoretical potential, there are currently few practical studies in the literature that examine the efficacy of deconstructed Halstead measures. As a result, a sizable gap remains, emphasizing the necessity for careful testing to assess the viability and effectiveness of such an advanced strategy. Closing this gap could lead to improved software engineering predictive tactics and a deeper understanding of software error prediction mechanisms. Our proposed solution not only provide an efficient mechanism for fault prediction in resource-constrained software system but also ensure consistency, which ultimately reduces the computational overhead of model.

1.8 Research Questions

RQ 1: How can the features (metrics) be decomposed in to more granular features to improve SFP?

Rationale: The purpose of this research question is to examine the decomposition of Halstead base metrics. These measures gauge a software program's complexity by counting the distinct operators and operands within it. They are essential in identifying implementation flaws and the amount of effort needed for development and upkeep. However, there are built-in drawbacks as well, such as the notion that all operators and operands are of equal significance and that all programming languages and development environments follow the same metrics. Thus, investigating possible techniques and approaches for dissecting Halstead's basic measurements into its component elements or sub-metrics is essential. This entails classifying arithmetic, logical, and control operators in addition to differentiating between input and output.

RQ 2: What is the impact of decomposed Halstead base metrics in SFP?

Rationale: The implications of using deconstructed Halstead base metrics in Software Fault Prediction (SFP) are examined in this research question. SFP comprises the use of machine learning and data analysis tools to proactively identify potential software flaws or vulnerabilities. This procedure helps programmers anticipate issues and address them before they become more serious, producing software that is more error-free and of higher quality. SFP, however, frequently ignores the subtleties of the algorithms and code that power system functionality. Comparing and evaluating the results obtained by breaking down Halstead base measurements with traditional measurements is crucial. To do this, it is necessary to choose pertinent performance measures, assess the impact of various datasets, and determine whether decomposed Halstead base metrics are appropriate in a variety of software engineering domains, such as reliability testing, prediction, and maintenance.

1.9 Research Scope

This research focuses on addressing the gap in literature by exploring the impact of decomposing Halstead metrics on the predictability of software fault prediction (SFP). The dissertation aims to evaluate how the decomposition of these metrics influences the accuracy and effectiveness of machine learning models in fault prediction. Specifically, the research will:

1.9.1 Focus Areas

Investigate the theoretical foundation and practical application of Halstead metrics in the context of SFP. Propose a method for decomposing Halstead metrics to extract potentially more informative features for machine learning models. Conduct experimental evaluations to assess the predictive performance of machine learning models using decomposed Halstead metrics.

1.9.2 Exclusions

The research will not explore alternative software metrics beyond the Halstead suite, as the study is confined to evaluating the decomposition of this specific metric suite. The research will not explore alternative software metrics beyond the Halstead suite, as the study is confined to evaluating the decomposition of this specific metric suite. Broader aspects of software fault prediction, such as runtime fault analysis, real-time fault detection, or non-machine-learning-based approaches, are outside the scope of this research. The deseration does not aim to propose new machine learning algorithms but will utilize existing algorithms to validate the effectiveness of Halstead metric decomposition.

1.9.3 Constraints and Limitations

The research is limited to publicly available datasets and does not include proprietary or domain-specific datasets due to accessibility constraints. The experimental results are based on controlled setups and may not account for all real-world complexities, such as rapidly evolving software ecosystems or diverse development methodologies. By clearly defining the boundaries, this research aims to provide a focused investigation into the role of Halstead metric decomposition in SFP, while acknowledging areas that remain outside its scope due to constraints and limitations.

1.10 Research Methodology

This study uses a systematic approach to enhance software fault prediction (SFP) through the use of machine learning methods using decomposed Halstead base metrics. Five interrelated phases make up the methodology, which is shown in Figure 1.2 of the dissertation and systematically fills in the gaps in conventional SFP approaches.

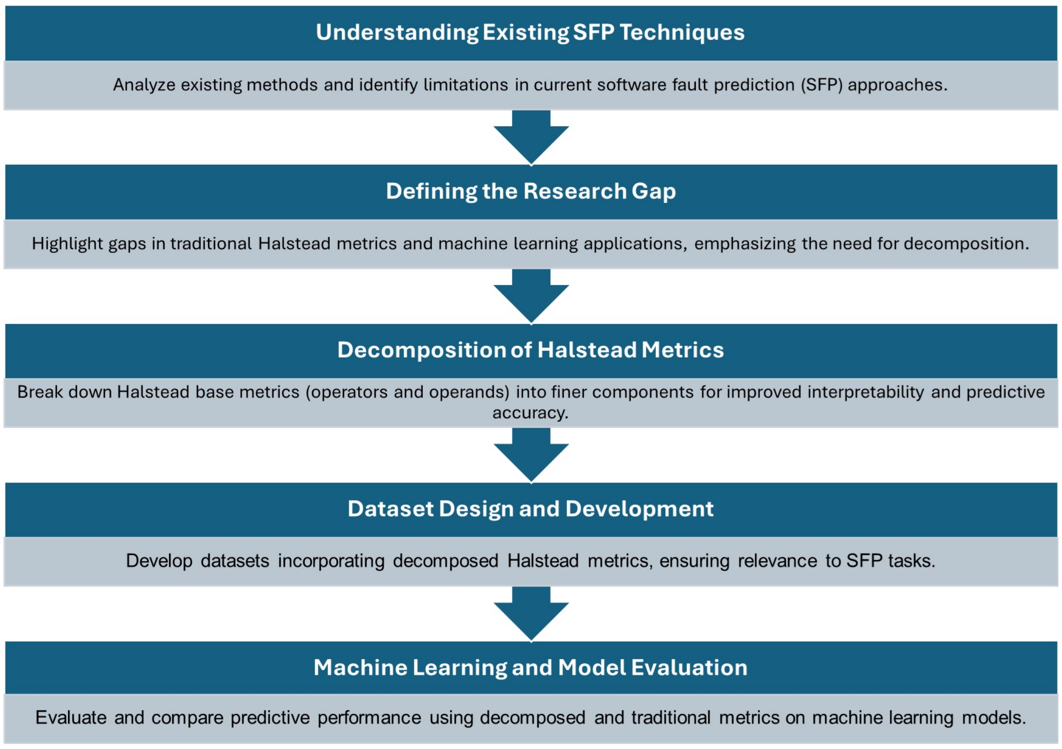
Understanding Existing SFP Techniques
Analyze existing methods and identify limitations in current software fault prediction (SFP) approaches.
Defining the Research Gap
Highlight gaps in traditional Halstead metrics and machine learning applications, emphasizing the need for decomposition.
Decomposition of Halstead Metrics
Break down Halstead base metrics (operators and operands) into finer components for improved interpretability and predictive accuracy.
Dataset Design and Development
Develop datasets incorporating decomposed Halstead metrics, ensuring relevance to SFP tasks.
Machine Learning and Model Evaluation
Evaluate and compare predictive performance using decomposed and traditional metrics on machine learning models.

FIGURE 1.2: Methodology of research

1.10.1 Understanding Existing SFP Techniques

A critical evaluation of previous research and approaches in software fault prediction is part of the first phase. This entails examining the constraints of Halstead metrics, evaluating their past implementations, and investigating the developments of SFP's machine learning algorithms. The review identifies shortcomings in conventional SFP models, especially the low granularity of Halstead metrics, which limits the predictive power of these models. This stage creates a strong basis for suggesting novel approaches by combining knowledge from earlier studies.

1.10.2 Defining the Research Gap

Building on the literature assessment, the study pinpoints a particular weakness: the low granularity of un-decomposed Halstead metrics, which limits their usefulness in fault prediction. In this stage, it is hypothesized that Halstead measures can be made more useful as predicting features by breaking them down into smaller parts. In order to better accommodate the complexity of contemporary software systems, the decomposition focuses on producing granular representations of operators and operands.

1.10.3 Decomposition of Halstead Metrics

Halstead base metrics must be systematically decomposed during this crucial stage. To improve their interoperability and predictive value, operators and operands are divided into subgroups.

Operators: Divided into assignment, arithmetic, logical, and relational categories.

Operands: Differentiated into variables and constants.

This decomposition facilitates the identification of fault-prone areas in software by capturing detailed interactions between code components. A custom-built metrics extractor is utilized to achieve this decomposition efficiently, ensuring accuracy and scalability.

1.10.4 Dataset Design and Development

A customized dataset is created for experimental assessment based on the decomposed Halstead metrics. This dataset consists of:

Decomposed Halstead metrics alongside traditional Halstead metric suite, Lines of Code (LoC) and McCabe complexity. Fault information obtained from software sources that are openly accessible. Testing of decomposed Halstead base metrics in real world settings is made possible by the dataset's assurance of comprehensiveness and relevancy. To preserve the contextual links between operators and operands, a hierarchical tree structure is used during the data extraction process.

1.10.5 ML Model Evaluation

The effectiveness of decomposed Halstead base metrics in SFP is assessed in the last stage using machine learning methods. Logistic Regression, Random Forests, Support Vector Machines (SVM), and Decision Trees are some of the models used. The created dataset is used to train and evaluate these models. To compare models and evaluate the advantages of decomposition, performance metrics like Accuracy, Precision, Recall, F-measure, and Area Under the Curve (AUC) are used. To guarantee resilience in a variety of software environments, sensitivity analyses are carried out. This thorough approach offers practical suggestions for enhancing SFP practices in addition to advancing the theoretical understanding of Halstead measures. The suggested method improves software engineering by resolving the drawbacks of conventional metrics and combining them with machine learning.

1.11 Contribution

As scholars, we undertake a critical investigation into the topic of software fault prediction (SFP) using Halstead metric suite and its essential function in improving processes related to software testing and quality assurance. Acknowledging the importance of this field, we conduct a thorough analysis of current SFP approaches, concentrating on the application of machine learning (ML) methods and various metrics as predictive markers. We attempt to contribute to this area by putting out a novel method for decomposing Halstead base metrics in the different types of operator and operands.

These line-level metrics are essential for measuring the size and complexity of code. We present an approach that first divides these metrics into operators and operands and then classifies them into different categories according to their attributes and functions. Using datasets created especially for this purpose, we do a thorough examination to verify the effectiveness of our suggested approach. We carefully compare the performance of our suggested approach with that of the original

Halstead base measures and other commonly used metrics in the SFP area by utilizing six ML classifiers.

We believe that this thorough investigation (Effectiveness of Decomposition of Halstead base metrics in SFP) will shed light on the innovative decomposition method's suitability and efficacy for software fault prediction jobs. Through a thorough analysis of our approach's performance in comparison to recommended practices and metrics, we believe in furthering the field of SFP approaches and providing practitioners with useful tools for improving software maintainability and accuracy.

Our work aims to close the knowledge gap between theoretical understandings and real-world implementations, which will ultimately promote stronger software development procedures and better software quality assurance systems.

1.12 Dissertation Outline

The dissertation outline is as follows: Chapter 2 briefly discusses the literature review. Chapter 3 focuses on the methodology. Chapter 4 elaborates the concept of decomposition and its implication in Halstead base metric. Chapter 5 elaborate the experimental design and results. Finally, Chapter 6 concludes the conclusion and future directions for research.

Chapter 2

Literature Review

Our main goal is to assess how well software metrics (features) can be decomposed to improve SFP through ML. In ML, feature decomposition is the process of dividing intricate features into smaller, more understandable parts. This method can lower dimensionality, improve model interpretability, and make it easier to find significant patterns in data. Models can increase prediction accuracy and generalization by better differentiating between pertinent and irrelevant information through feature decomposition. Although feature decomposition has been shown to be useful in a number of fields [70–72], its use in SFP has not yet been investigated. We have undertaken a thorough review from two different angles to fill the vacuum in the literature on this subject.

Firstly, we investigated prevailing trends in SFP, with a specific focus on the utilization of Machine Learning (ML) algorithms and datasets. This exploration not only provided us with insights into the current state of SFP but also guided the selection of appropriate datasets and ML algorithms for our experimental investigations.

Secondly, we aimed to identify commonly employed software metrics used in conjunction with the Halstead metrics suite for SFP. This objective was pivotal, as SFP often necessitates the integration of diverse software metrics with ML techniques for accurate prediction of software faults. By identifying these commonly used software metrics and understanding their relationship with the Halstead metrics suite, we aimed to identify a robust decomposition of features for SFP that incorporates a comprehensive range of factors for improved fault prediction accuracy.

Through our literature review, we laid the groundwork for our subsequent empirical analyses, facilitating a deeper understanding of the complexities inherent in software fault prediction and potential chances for improvement.

2.1 Prevailing Trends in SFP

We meticulously examined a comprehensive selection of systematic literature reviews (SLRs) to swiftly discern and synthesize the prevailing trends in SFP. Through this meticulous process, we aimed to gain a panoramic view of the current landscape in SFP, paying particular attention to the evolving nature of datasets and the dynamic utilization of a diverse array of Machine Learning (ML) algorithms. By harnessing the insights gleaned from these SLRs, we were able to ascertain the predominant research directions and methodologies, facilitating the identification of the most pertinent and contemporary research gaps in the field of SFP.

Moreover, this thorough examination of systematic literature reviews served as a robust foundation for the subsequent steps of our research, enabling us to make informed decisions regarding the selection of relevant datasets and the adoption of suitable ML algorithms for our experimental investigations. This strategic approach not only bolstered the credibility of our research but also fortified the rigor of our methodology, ensuring that our subsequent analyses and experiments were firmly grounded in the most up-to-date and relevant research findings.

By leveraging the comprehensive insights garnered from literature review of SLRs, we were able to lay the groundwork for a well informed and comprehensive analysis of the decomposed Halstead base metrics in the context of SFP. SLRs findings equipped us with a holistic understanding of the current research landscape, enabling us to navigate the complexities of SFP research with precision and purpose, and positioning our study to make significant contributions to the advancement of fault prediction methodologies in software engineering.

Catal and Diri [38] emphasizes several key findings in SFP research, including the widespread utilization of method-level metrics, the growing prevalence of public datasets, and the increased adoption of machine learning techniques. These observations serve as important insights for advancing the field and enhancing the precision and efficacy of SFP models. By recognizing and implementing these recommendations, researchers and practitioners can contribute to the ongoing development and refinement of SFP methodologies.

Catal [15] presents a comprehensive survey of the software engineering literature on SFP, covering both machine learning-based and statistical-based approaches. The survey findings indicate that a significant proportion of the studies examined in this survey concentrate on method-level metrics, with machine learning techniques being the primary approach employed for constructing prediction models. Notably, the study suggests that Naïve Bayes emerges as a robust machine learning algorithm suitable for supervised SFP.

Hall et al [73] highlights the presence of exemplary fault prediction studies while emphasizing the existence of unresolved inquiries concerning the development of efficient fault prediction models for software systems. It asserts the necessity for additional studies that conform to reliable methodologies and consistently document contextual details and methodologies. The accumulation of a larger body of such studies would facilitate meta-analysis, provide practitioners with the confidence to adeptly choose and implement models in their systems, and ultimately augment the influence of fault prediction on the quality and cost of industrial software systems.

Malhotra [74] review that focuses on evaluating the performance of machine learning (ML) techniques in Software SFP. The review involves analyzing the quality of 64 primary studies conducted between 1991 and 2013. The characteristics of these studies, including metrics reduction techniques, metrics used, data sets, and performance measures, are summarized. The performance of ML techniques in SFP is assessed by comparing them to models predicted using logistic regression.

Furthermore, the performance of ML techniques is analyzed in comparison to other ML approaches.

Wahono [75] analyzes 71 studies published between 2000 and 2013 to understand trends, datasets, methods, and frameworks used in software fault prediction. The research primarily focuses on estimation, association, classification, clustering, and dataset analysis. Classification methods dominate the studies, accounting for 77.46%, followed by estimation methods at 14.08%, and clustering/association methods at 1.41%. Public datasets are utilized in 64.79% of the studies, while private datasets are used in 35.21%. The review identifies seven frequently employed methods: Logistic Regression, Naïve Bayes, K-Nearest Neighbor, Neural Network, Decision Tree, Support Vector Machine, and Random Forest.

Rathore and Kumar [21] provides an examination of SFP through a comprehensive analysis of the existing literature. The review encompasses various aspects including software metrics, fault prediction techniques, concerns related to data quality, and evaluation measures for performance. By exploring these domains, the review sheds light on the challenges and methodological issues that are inherent in this field. Existing studies predominantly concentrate on object-oriented (OO) metrics and process metrics, and they primarily utilize publicly available data. Statistical techniques, particularly binary class classification, are widely employed in these studies. A considerable amount of attention has been given to tackling issues such as high data dimensionality and class imbalance quality. Performance metrics like accuracy, precision, and recall are commonly used in assessing the performance of these fault prediction techniques.

Caulo [76] introduces a comprehensive taxonomy of metrics for SFP. The taxonomy comprises a total of 526 metrics employed in research papers published from 1991 to 2017. The paper emphasizes the significance of evaluating the efficacy of each metric in SFP. Additionally, the author proposes to categorize the identified metrics based on their co-linearity, thereby facilitating the exploration of relationships between different metrics and their collective influence on SFP.

Pandey et al [40] presents a comprehensive analysis of machine learning-based methods for SFP. Its primary objective is to explore and summarize the current state of the field by examining a diverse range of ML techniques and approaches utilized in SFP. The survey underscores the importance of leveraging machine learning for fault prediction and acknowledges its potential to improve software quality and reliability.

It delves into various machine learning algorithms, including decision trees, support vector machines (SVM), neural networks, and ensemble methods, and investigates their specific applications in the context of fault prediction.

Pachouly et al [57] provides a comprehensive overview of software fault prediction using artificial intelligence (AI). It covers four key aspects: datasets, data validation methods, approaches, and tools. The review emphasizes the importance of high-quality datasets and explores different validation methods to ensure accurate and reliable data.

It discusses various AI techniques and algorithms used in defect prediction, highlighting their strengths and limitations. Additionally, it identifies and examines tools and frameworks that aid in implementing and evaluating AI models for defect prediction.

A large number of Systematic Literature Reviews (SLRs) in the field of Software Fault Prediction (SFP) have produced valuable insights and suggestions that have advanced this field considerably. The studies that are part of these evaluated SLRs have carefully examined software fault prediction performance, providing insight into the effectiveness of different techniques and strategies. As seen in Figure 2.1 of the examined SLRs, the distribution of these studies over time shows how the field of interest in software fault prediction has changed over time. The figure provides a thorough summary of this distribution, highlighting changes and patterns in the attention and emphasis of research.

Surprisingly, since 2005, there has been a discernible rise in the number of studies published, suggesting that there is an increasing focus on current and relevant research in this area. This increase is in line with the creation of the PROMISE repository in 2005, which gave academics access to publicly available datasets for testing and validation.

4.png 4.png 4.png 4.png 4.png 4.png 4.png 4.png 4.png

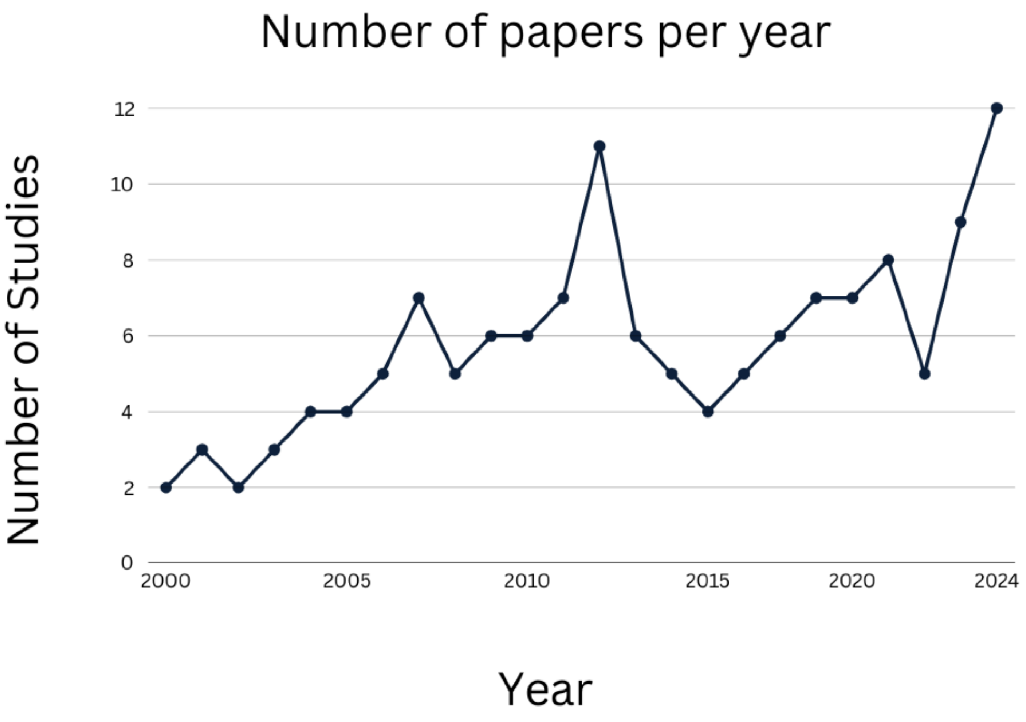

Figure 2.1: Number of SLRs on SFP per year

In conclusion, the examined SLRs provide insightful information about the development and course of software fault prediction research, emphasizing the field's continuous quest of innovation and advancement in software engineering. By means of a methodical analysis and integration of extant literature, these reviews enhance our comprehension of cutting-edge approaches, patterns, and obstacles in software fault prediction. This, in turn, directs future research paths and shapes industry standards for the improvement of software development procedures and results.

According to the conclusions drawn from the primary papers that were included in the Systematic Literature Review (SLR) that was examined, some journals that are dedicated to software fault prediction are very important in the field. These results, which were carefully extracted from the academic literature, are briefly summarised in Figure 2.2, which explains the importance of particular publications in the field of software fault prediction.

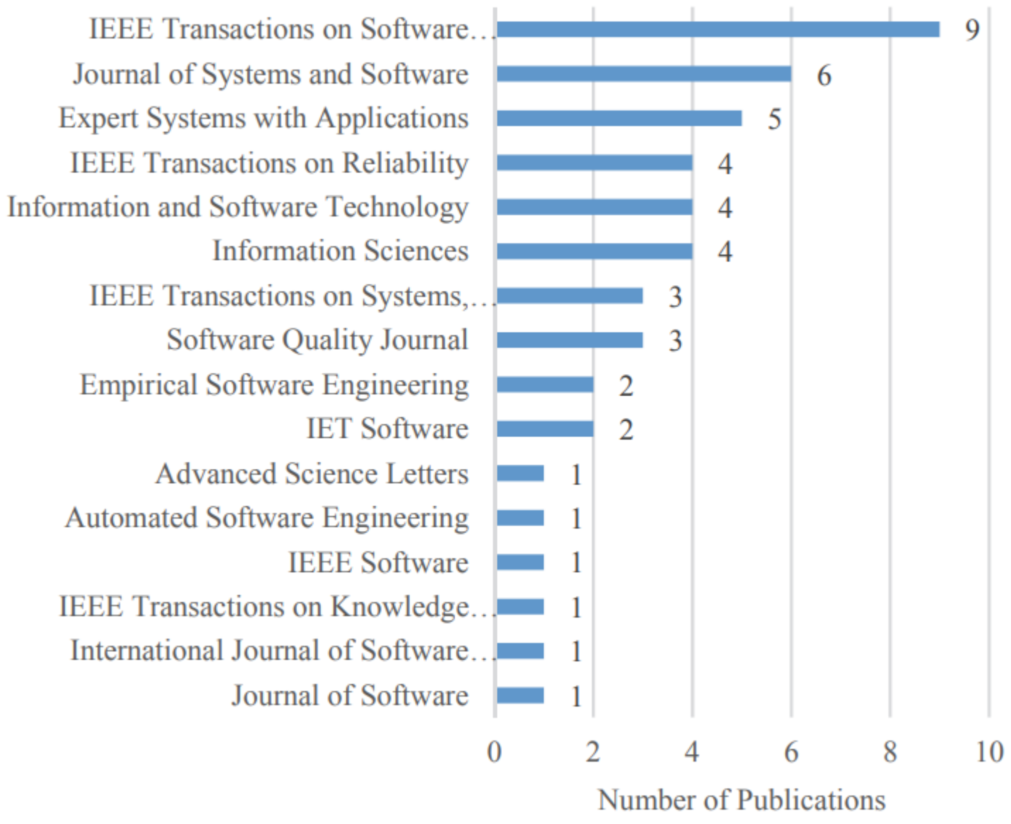

FIGURE 2.2: Journal publication and distribution of studies in SLRs

The purposeful omission of conference proceedings from the graph's portrayal is noteworthy. This limiting step makes sure that the analysis is only about the contributions of journals in this particular context, which improves the analysis's relevance and clarity.

The graphical representation, which is based on the isolation of journal publications, offers a thorough overview of the scholarly landscape and enables a nuanced comprehension of the primary sources that propel breakthroughs and insights in the field of software fault prediction research.

The variety of dataset types used in software fault prediction research between 1990 and 2023 is shown in Figure 2.3. According to the analysis, 52% of the research projects that were analysed depended on public databases, whereas 31%

of the studies used private datasets. This represents a considerable majority. The majority of public datasets come from repositories such as NASA MDP (Metrics Data Programme) and PROMISE, where they are freely accessible to the larger research community.

Private datasets, on the other hand, are the intellectual property of businesses and are not publicly shared like their public equivalents. Private datasets are typically restricted to use within their own firms and cannot be accessed for external scrutiny or analysis. As a result, different methods must be used to validate and replicate research findings. The distribution throughout the years shows how interest in different dataset types changes over time.

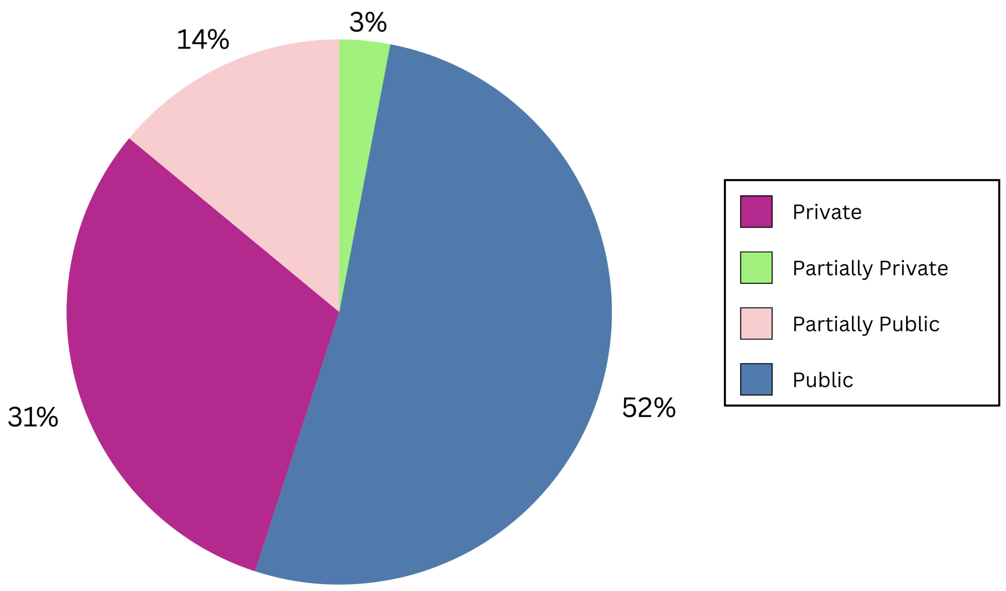

FIGURE 2.3: Type of datasets in SFP

Regretfully, a considerable fraction of the research, at 31% of the total, made use of private datasets.

According to this statistic, just one study out of every three produces findings that are comparable and repeatable. However, because their datasets are not publicly available, it is impossible to compare the results of this research with those of suggested models. Notably, as [77] emphasizes, the use of standard datasets makes research projects reproducible, disputable, and verifiable.

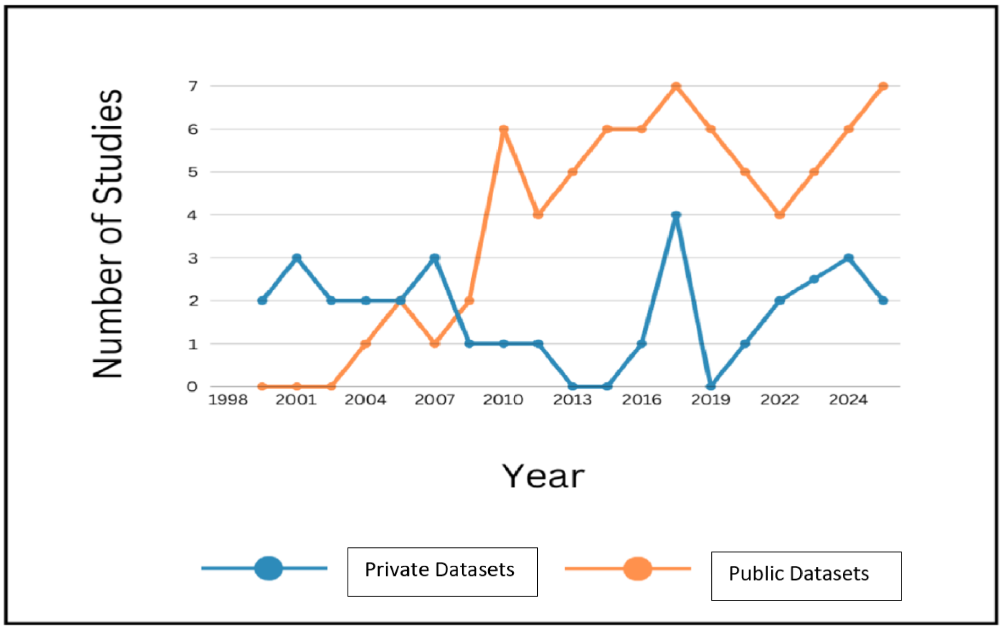

FIGURE 2.4: Private, Public dataset trend over the time distribution.

TABLE 2.1: Public and Private datasets with their names and sours

Dataset Name	Type	Location/ Source
PROMISE Dataset	Public	http://promise.site. uottawa.ca/SERepository
NASA MDP (Metric Data Program)	Public	https://mdp.ivv.nasa.gov
Apache Dataset	Public	https://github.com/apache
Proprietary Software X	Private	Confidential – Internal Repository
Proprietary Software Y	Private	Confidential - Client Dataset

The distribution of primary research by source and over time is shown in Figure 2.4. Interestingly, since 2003, there has been a noticeable increase in publications and the use of public datasets for research on software fault prediction. The PROMISE repository was established in the same year as this trend. Moreover, scholars are becoming increasingly aware of the need to utilize public datasets in their studies.

Some of the public and private datasets are listed in Table 2.1 with their sours information.

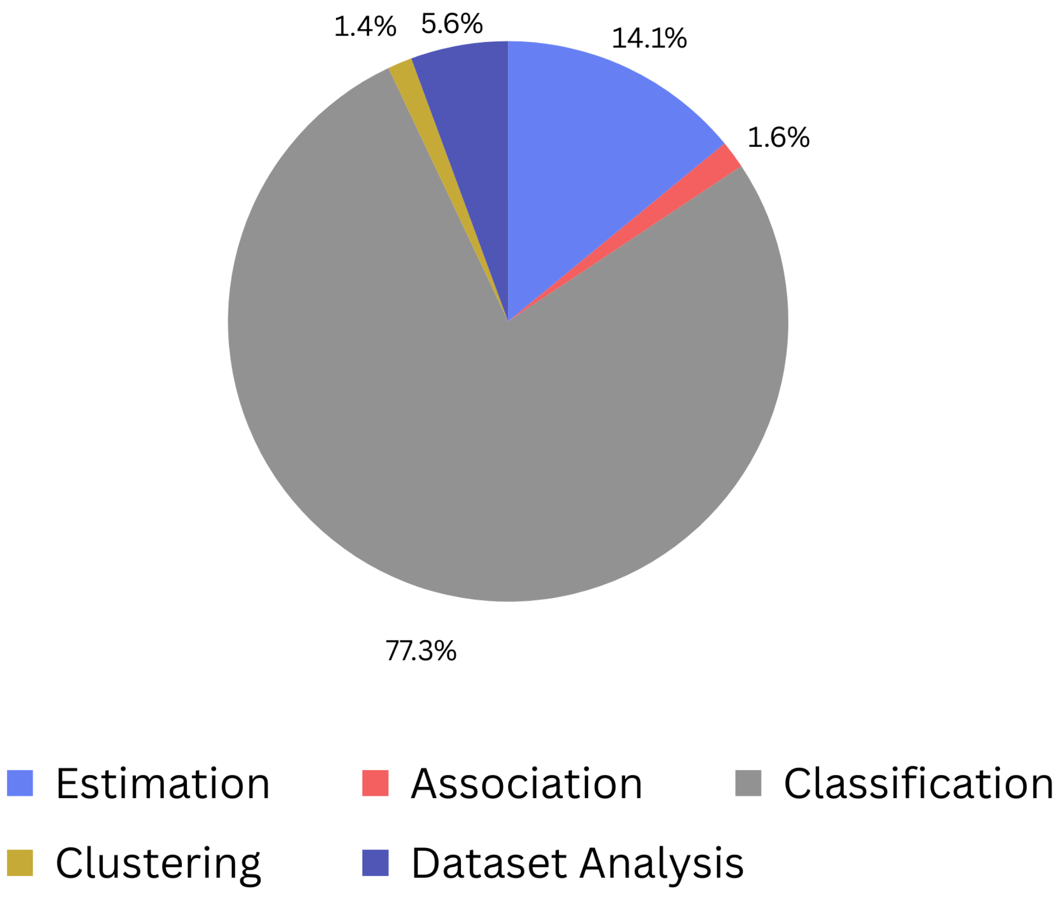

FIGURE 2.5: Research dimensions in the domain of SFP

A detailed breakdown of research areas related to software fault prediction from 2000 to 2024 is shown in Figure 2.5. The analysis shows that classification issues account for a preponderant 77.3% of research efforts. This large percentage emphasizes how critical it is to correctly classify software fault in both academic and professional settings.

On the other hand, a lesser but significant portion of 14.1% of the research addresses estimating methods, emphasizing how important it is to forecast the frequency and seriousness of fault in software systems.

Moreover, a small percentage of original research 5.6% focuses on themes related to dataset analysis. The focus on dataset analysis highlights the importance of feature engineering to enhancing the performance of defect prediction models. Interestingly, even though they are important, clustering and association are only covered by 1.4% and 1.6% of all studied subjects. Even though they are less common, these research areas provide insightful information about different strategies and techniques for dealing with software fault prediction problems.

To sum up, Figure 2.5 explanation of the research dimensions distribution highlights the wide range of approaches and strategies used to achieve accurate software fault prediction. Through exploring a wide range of subjects related to association, clustering, dataset analysis, estimation, and classification, researchers aim to promote innovation and push the boundaries of this vital field of software engineering.

It can be concluded that a predominant portion of software fault prediction researchers have gravitated towards the classification paradigm as their primary research focus. This preference can be attributed to several factors, each contributing to the prominence of classification in the field of SFP.

Firstly, the alignment between classification topics and industrial demands underscores its significance. Industrial requirements necessitate methods capable of accurately predicting which modules are more susceptible to defects, facilitating the allocation of testing resources with greater precision.

By employing classification techniques, researchers can provide actionable insights into defect-prone areas, thereby aiding in the optimization of testing strategies and resource allocation within software development projects.

Secondly, the widespread availability and utilization of datasets, such as the NASA MDP dataset, predominantly designed for classification tasks, serve as another catalyst for the emphasis on classification methods. These datasets offer comprehensive repositories of historical software metrics and defect data, facilitating the exploration and application of various classification algorithms for predictive modeling purposes.

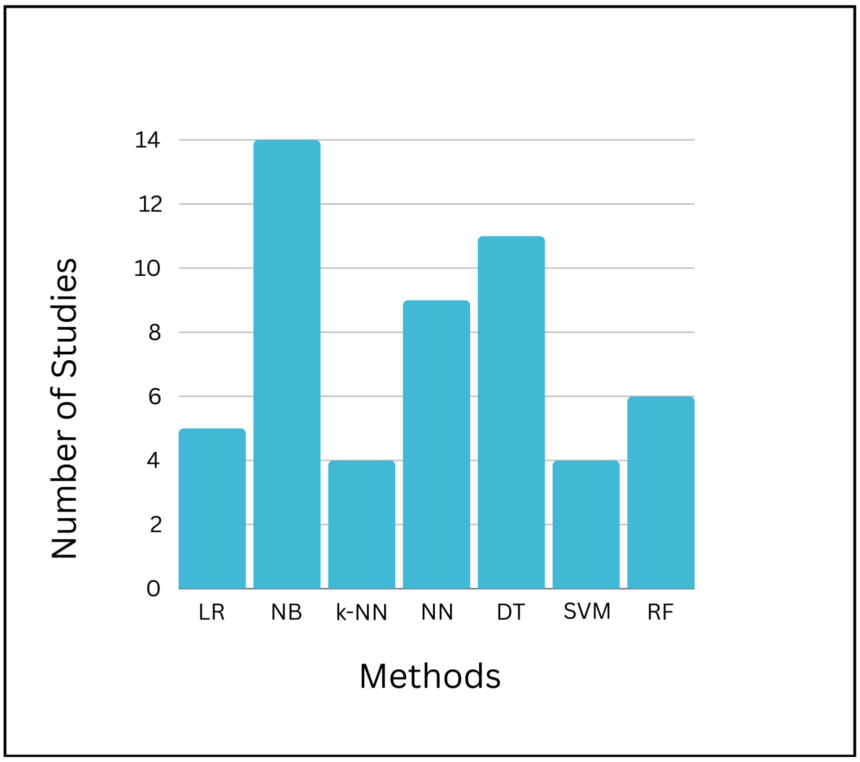

Figure 2.6: Most used methods in SFP

However, a notable dearth of studies in clustering and association related topics persists within the realm of software fault prediction research. This dearth can potentially be attributed to the perceived inadequacies of clustering and association methods in yielding desirable performance outcomes. Unlike classification techniques, clustering and association methods may encounter challenges in effectively delineating distinct groups or uncovering meaningful associations within software development data. Consequently, researchers may be disinclined to pursue these avenues due to concerns regarding the viability and publish ability of research outcomes that fail to meet established performance benchmarks. In essence, while classification remains the predominant focus in software fault prediction research, the delineated reasons underscore the multifaceted dynamics shaping research priorities within the field. The convergence of industrial imperatives, dataset availability, and performance considerations collectively influence the research landscape, ultimately guiding the selection and exploration of research topics within software fault prediction.

In the realm of SFP, classification stands as the prevailing method employed for predictive modeling. Given the significance of classification techniques in this domain, it is imperative to discern the prominent methodologies utilized for software fault prediction. As such, a comprehensive examination has been undertaken to identify the seven most applied classification methods in software fault prediction, delineating their significance and prevalence within the field of SFP. These methods, elucidated in Figure 2.6, encompass a spectrum of algorithmic approaches tailored to address the intricacies inherent in software fault prediction.

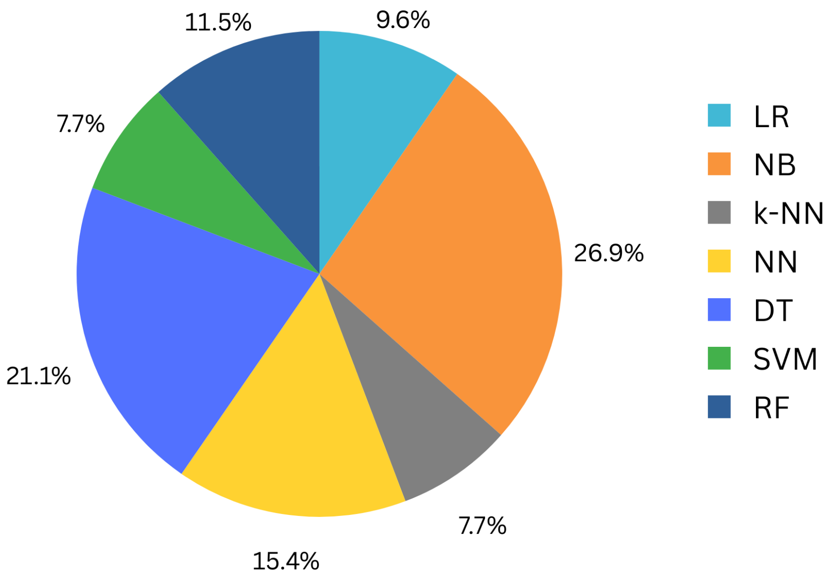

FIGURE 2.7: Methods distribution in studies

In elucidating these seven predominant classification methods, the foundational pillars of software fault prediction are fortified, providing practitioners and researchers with invaluable insights into the diverse array of methodologies available to address the challenges inherent in predicting software faults.

Naive Bayes (NB), Decision Trees (DT), Neural Networks (NN), and Random Forests (RF) represent the four most commonly employed machine learning algorithms in the context of software fault prediction.

Remarkably, these algorithms were embraced by a significant majority, constituting approximately 75% of the studies analyzed, as depicted in Figure 2.7. This prevalence underscores the widespread recognition and utilization of these algorithms within the software engineering community for the critical task of fault prediction.

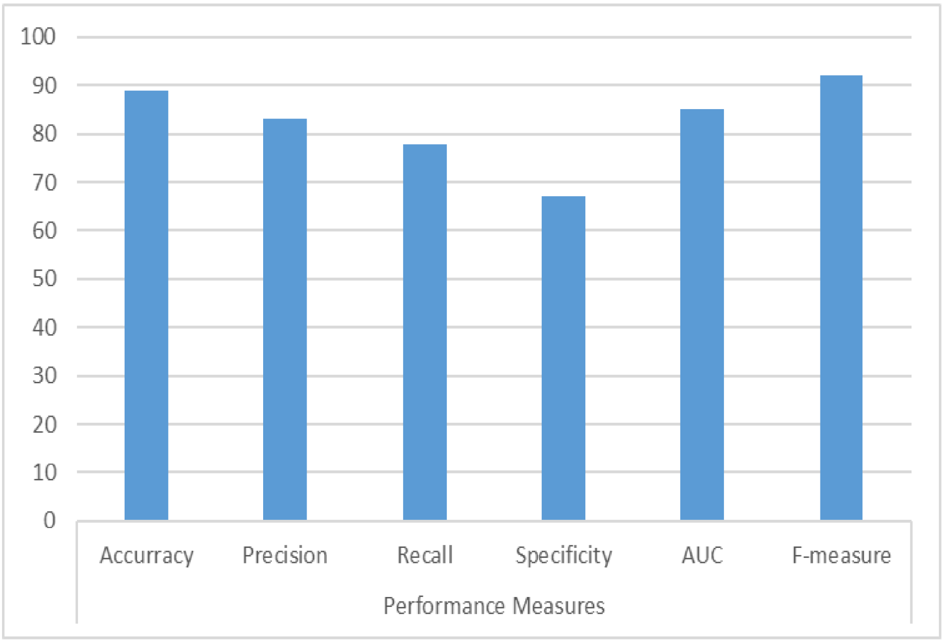

FIGURE 2.8: Performance measure used in studies.

Figure 2.8 illustrates the spectrum of performance measurements employed across numerous studies within the Software Fault Prediction (SFP) discipline. It is evident that among the array of performance metrics, Recall and Receiver Operating Characteristics (ROC) have garnered considerable attention and are prominently featured in the literature. Conversely, the category denoted as ”Miscellaneous” encompasses a collection of evaluation measures that have been infrequently utilized in SFP research, as depicted in Figure 2.8. Additionally, Precision and Accuracy emerge as significant contributors to the evaluation landscape of SFP, finding consistent application across a majority of studies within the field. The adoption of these metrics underscores their relevance and utility in assessing the predictive capabilities of machine learning algorithms for software fault detection. Conversely, certain performance metrics such as F-measure, Probability of False Alarm (PFA), Specificity, Balance, G-mean, Completeness, and False Negative Rate (FNR) are observed to be less prevalent in the literature, as indicated by their minimal representation in Figure 2.8. Despite their potential value in providing nuanced insights into model performance, their comparatively limited utilization suggests a need for further exploration and validation within the context of software fault prediction.

In summary, Figure 2.8 serves as a comprehensive visualization of the distribution of performance measurements across SFP studies, highlighting the varying degrees of emphasis placed on different performance metrics and signaling potential areas for future research inquiry and methodological refinement. Our primary focus was on Machine Learning (ML) however, Deep Learning (DL) also proves to be effective for large datasets when sufficient computational resources are available for Software Fault Prediction (SFP) [78].

2.2 Software Metrics

Software fault prediction relies on a wide range of metrics to assess the quality and reliability of software code [42]. These metrics, categorized at different levels of granularity within the software hierarchy, provide insights into potential fault-prone areas. This section explores software metrics used in fault prediction at various levels: file, class, method, component, and line [79].

2.2.1 File level metrics

Provide a high-level overview of individual source code files and their characteristics. Common file-level metrics include:

Lines of Code (LOC): Measures the number of lines of code within a file.

Cyclomatic Complexity: Quantifies the structural complexity of a file based on control flow.

Code Duplication: Identifies duplicated code segments within files.

Halstead Complexity Measures: Evaluate the volume and difficulty of code within a file.

Analyzing file-level metrics helps identify files that may be more prone to faults due to their complexity or size.

2.2.2 Class level metrics

Focus on object oriented programming constructs and assess the characteristics of individual classes or modules. Key class-level metrics include:

Coupling: Measures the degree of interdependence between classes.

Cohesion: Indicates the strength of the relationships within a class.

Depth of Inheritance Tree (DIT): Represents the number of classes in the inheritance hierarchy.

Identifying classes with high coupling or low cohesion can highlight potential areas of concern for fault prediction.

2.2.3 Method Level Metrics

Delve into the characteristics of individual methods or functions within classes. These metrics include:

Cyclometic Complexity: Evaluate the complexity of individual methods based on control flow.

Lines of Code (LOC) per Method: Measures the size of individual methods.

Halstead base metrics: Measures unique and distinct operators and operands of individual methods.

Analyzing method-level metrics helps pinpoint specific methods that may be prone to faults due to their complexity or length.

2.2.4 Component Level Metrics

Assess the characteristics and interactions of larger software components or subsystems. These metrics include:

Component Coupling: Measures the degree of interdependence between components.

Component Cohesion: Indicates the strength of the relationships within a component.

Component Size: Quantifies the size or complexity of individual components.

Halstead base metrics: Measures unique and distinct operators and operands of individual components.

Identifying components with high coupling or low cohesion can aid in predicting faults within complex software systems.

2.2.5 Line Level Metrics

Provide granular insights into individual lines of code within files. Common line level metrics include:

Code Churn: Measures the frequency of changes to individual lines of code.

Comment Density: Quantifies the proportion of comments to code within a line.

Halstead base metrics: Measures unique and distinct operators and operands of individual lines of code.

Analyzing line-level metrics helps identify specific lines of code that may be more prone to faults due to frequent changes or lack of documentation.

Software development teams can efficiently detect and prioritize potentially fault-prone areas within the code base by utilizing software metrics at various degrees of granularity. These metrics support proactive fault prediction and mitigation efforts by offering insightful information on the complexity, size, cohesiveness, and coupling of software components. Measuring individual functions or methods is an example of granular metrics that can be used to identify particular locations where errors may arise due to complexity or size. Higher-level metrics can highlight more extensive structural problems that may affect maintainability and reliability. Examples of these metrics are those that evaluate the overall system design or the inter-module relationships. To get useful insights for software fault prediction and prevention, these metrics must be interpreted in combination with contextual information and domain expertise. The context needed to comprehend why some metrics could point to a problem in one circumstance but not another is provided by domain knowledge. For example, in a performance-critical module where optimization is crucial, a high Cyclometic complexity might be acceptable, but in other areas of the system, it might be cause for concern.

Furthermore, the interpretation of metrics must to take into account contextual elements including the team's experience, the development environment, and previous defect data. Teams can improve the accuracy of fault prediction models by correlating particular metrics with previous faults using historical data, which can be especially useful. Furthermore, knowing how external factors—like changing project requirements or technology limitations—affect the data might aid in making better-informed judgments. As a result, even if software metrics are effective in pointing out possible trouble spots, their full potential is only fully realized when paired with contextual knowledge and domain-specific insights. This all-encompassing strategy guarantees that development teams may continuously enhance the overall quality and resilience of the programme in addition to more successfully anticipating and preventing errors.

2.3 Software Metrics with Halstead in SFP

Our thorough investigation of studies in the domain of Software Fault Prediction (SFP) has been specifically geared toward identifying research endeavors that have exclusively integrated the Halstead metric suite as a fundamental component within their metric set for software fault prediction. By focusing on these specialized studies, we aimed to discern the distinct impact and significance of the Halstead metric suite in isolation, thereby elucidating its individual contributions to the realm of fault prediction within the software engineering domain. Through this meticulous analysis, we sought to unravel the unique insights and implications derived from the exclusive utilization of the Halstead metric suite in the context of SFP. By delving into the methodologies, findings, and limitations of these specific studies, we aspired to gain a nuanced understanding of the inherent strengths and potential limitations of the Halstead metrics when employed as the primary predictive framework for software fault prediction.

Furthermore, by scrutinizing studies that exclusively incorporate the Halstead metric suite, we aimed to delineate the extent to which these metrics contribute to the overall efficacy and accuracy of fault prediction models. This exploration has enabled us to identify the specific nuances and intricacies associated with the integration of the Halstead metric suite in the predictive analytics framework, thereby paving the way for a comprehensive evaluation of its role and significance in the broader context of software fault prediction. By synthesizing the findings from these studies, our research endeavors to provide a comprehensive and nuanced perspective on the exclusive utilization of the Halstead metric suite in SFP, thereby offering valuable insights into its unique contributions and potential applications in the development of robust and effective fault prediction models in software engineering. The articles to find out the accompanying metrics of Halstead metrics suite in SFP have been discussed below along with the summary in Table 2.2.

Chiu [62] reported the classification accuracy of Halstead, when used with McCabe, LoC, and Branch count. The modeling has been performed using four different classification algorithms, i.e., LR, SVM, ANN, and DIN The experiment on KC2 dataset shows the best results when used IDN for modeling.

Dejaeger et al [63] includes LR, RF, and Bayesian Network (BN) classifiers for modeling on 11 public datasets. Halstead metrics suite along with McCabe and LoC has been used as an Independent variable (IV) The results, both in terms of the AUC and H-measure have been recorded wherein NB outperforms.

Table 2.2: Summarized view of studies using Halstead metric suite

Article	**Metrics**	**Dataset**	**Technique**	**Performance Measure**	**Contribution**	**Best Results**
Chiu. (2011)	Halstead, McCabe, LOC, Branch Count	KC2	LR, SVM, ANN, Integrated decision network approach (IDN)	Acc, Pre, Recall, F-measure	Compare different ML Algorithm	IDN Acc(.87)
Dejaeger *et al.* (2013)	Halstead, McCabe, LOC	JM1, KC1, MC1, PC1, PC2, PC3, PC4, PC5, EC12.0a, EC12.1a, EC13.0a	LR, RF, NB	AUC, H-Measure	Compare different ML Algorithm	NB AUC(.85)
Arar and Ayan. (2015)	Halstead, McCabe	KC1, KC2, JM1, PC1, CM1	ANN, Artificial Bee Colony (ABC)	AUC, Acc	Compare different ML Algorithm	ANN AUC(.79)

Continued on next page

Article	Metrics	Dataset	Technique	Performance Measure	Contribution	Best Results
Dhanajayan and Pillai. (2017)	Halstead, McCabe, LOC, Branch Count	CM1	NB, RF, ANN, Spiral life cycle model-based Bayesian classification (SLMBC)	False Negative Rate, False Positive Rate, Overall error rate	Compare different ML Algorithm	RF Acc(.85)
Bhandari and Gupta. (2018)	Halstead, McCabe, LOC	JM1, PC1, KC1, jEdit	RF, DT, NB, SVM, ANN	Acc, F1-Score, precision, recall, AUC	Compare different ML Algorithm	SVM F1-score(.89)
Shippey *et al.* (2019)	Halstead, McCabe, LOC, Branch Count	T2, T1, EJDT, ArgoUML, AspectJ, JMOL, GenoViz	NB, DT, RF.	Recall, Pre	Compare different ML Algorithm	SVM Acc(.82)
Ahmed *et al.* (2020)	Halstead, McCabe, LOC, Branch Count, Call Pairs	PC1, PC2, PC3, PC4, PC5, JM1, KC1, MC1, Ecl2.0a, Ecl2.1a, Ecl3.0a,	DT, NB, SVM, RF, KNN, LR,	AUC	Compare different ML Algorithm	SVM Acc(.86)

Continued on next page

Article	Metrics	Dataset	Technique	Performance Measure	Contribution	Best Results
Cetiner and Sahingoz. (2020)	Halstead, McCabe, LOC	PC1, JM1, KC1, CM1, KC2	DT, NB, KNN, SVM, RF, MLP, Extra Trees, Ada boost, Gradient Boosting, Bagging	Acc	Compare different ML Algorithm	RF Acc(.88)
Kulamala *et al.* (2021)	Halstead, McCabe, LOC	PC1, PC2, PC3, PC4, CM1, JM1	LR, NB, D,T MLP, SVM, RF, LSSVM	Acc, AUC, F1-Score	Compare different ML Algorithm	LSSVM Acc(.88)
Liu et al. (2022)	Halstead, McCabe, LOC	PC1, PC2, PC3, PC4, CM1, JM1, KC3 DT, CRV, BN, LS, LR	LR, NB, D,T MLP, SVM, RF, LSSVM	Acc, AUC, F1-Score	Compare different ML Algorithm	SVM Acc(.89)
Maria *et al.* (2023)	Halstead, McCabe, LOC	CM1, JM1, KC1	NB	Acc, Recall, Precision, F1-Score	Compare different ML Algorithm	NB Acc(.87)

Continued on next page

Article	**Metrics**	**Dataset**	**Technique**	**Performance Measure**	**Contribution**	**Best Results**
Susmita and Luiz (2024)	Halstead, McCabe, LOC	PC1, CM1, JM1, KC2, KC1	SVM, RF, K-nn, NN	Acc	Compare different ML Algorithm	RF Acc(.87)

Concluded

Arar and Ayan [64] utilized artificial neural networks (ANN) and the ABC optimization algorithm to analyze five datasets from the NASA Metrics Data Program repository. The classification approach was evaluated based on several performance indicators, including accuracy, probability of detection, probability of false alarm, balance, Area Under Curve (AUC), and Normalized Expected Cost of Misclassification (NECM). Halstead and McCabe metrics were employed as independent variables (IV). The experimental findings demonstrated the successful creation of a cost-sensitive neural network through the application of the ABC optimization algorithm.

Dhanajayan and Pillai [65] assess the SFP capability of Halstead, McCabe, LOC, and Branch Count on CM1 data set using NB, RF, ANN, Spiral life cycle model-based Bayesian classification (SLMBC). The performance has been evaluated using False Negative Rate, False Positive Rate, and Overall error rate.

Bhandari and Gupta [66] proposes a spiral life cycle model-based Bayesian classification technique for efficient SFP and classification. In this process, initially, the independent software modules are identified which are Halstead, McCabe, and LoC. The experiment results show that RF achieves higher accuracy, precision, recall, probability of detection, F-measure, and lower error rate than the rest of the techniques.

Shippey et al [67] employed the utilization of Abstract Syntax Tree (AST) n-grams to detect characteristics of faulty Java code that enhance the accuracy of defect prediction. Various metrics such as Halstead, McCabe, Lines of Code (LoC), and Branch Count have been applied to train Naïve Bayes, J489, and Random Forest models. The outcome reveals a strong and statistically significant correlation between AST n-grams and faults in certain systems, demonstrating a substantial impact.

Ahmed et al [80] proposed a software fault predictive development model using machine learning techniques that can enable the software to continue its projected task. Halstead, McCabe, LoC, Branch count and Call pairs have been used for modeling SVM, DT, NB, RF, KNN, and LR on three defect datasets in terms of f1 measure. The experiment results are in favor of LR.

Cetiner and Sahingoz [68] conducted a comparative analysis of machine learning-based software fault prediction systems by evaluating 10 learning algorithms including Decision Tree, Naïve Bayes, K-Nearest Neighbor, Support Vector Machine, Random Forest, Extra Trees, Ada boost, Gradient Boosting, Bagging, and Multi-Layer Perceptron. The analysis was performed on the public datasets CM1, KC1, KC2, JM1, and PC1 obtained from the PROMISE warehouse. Halstead, McCabe, and LoC were utilized for modeling the classification algorithms. The experimental findings demonstrated that the Random Forest (RF) model exhibited favorable accuracy levels in software fault prediction, thus enhancing the software quality.

Kumar et al [69] aimed to create and compare different SFP models using Least Squares Support Vector Machine (LSSVM) with three types of kernels: Linear, Polynomial, and Radial Basis Function (RBF). ***Maria et al*** [81] explores the application of Bayesian Networks for software defect prediction to enhance prediction robustness and interpretability. The study evaluates three algorithms for constructing Bayesian Networks: K2, Hill Climbing, and Tree Augmented Naive Bayes (TAN), and compares their performance against Decision Tree and Random Forest classifiers. The authors use three datasets (CM1, JM1, and KC1) from the PROMISE repository, leveraging McCabe and Halstead complexity metrics. Performance evaluation employs cross-validation with metrics including Accuracy, Recall, and F1 Score. The study finds that while Bayesian Networks exhibit lower variability across folds, their overall predictive performance is competitive, providing more stable classification results compared to traditional methods.

Susmita and Luiz [82] presents software fault prediction models emphasizing interpretability using machine learning techniques. The study uses five datasets from the PROMISE repository: cm1, kc1, kc2, jm1, and pc1. Performance metrics employed include Accuracy, F1 Score, and AUC. The models were developed using Support Vector Machines (SVM), k-Nearest Neighbors (KNN), Random Forest (RF), and Artificial Neural Networks (ANN). To enhance model transparency, Local Interpretable Model-Agnostic Explanations (LIME) and SHapley Additive exPlanations (SHAP) were used.

These studies focus on identifying software modules as either faulty or non-faulty by employing several software metrics, such as Halstead software metrics, McCabe, and Lines of Code (LoC). Multiple machine learning models are studied and assessed on a varied selection of open source projects. The performance of these models is tested using popular measures, communally including Accuracy, F-measure, and AUC. The results show that some models perform better than others in terms of classification efficacy, with performance varying based on the different datasets. Overall, the findings indicate that when machine learning models assessed, Random Forests is optimized [81], they typically perform better in software fault prediction, particularly when assessed using Accuracy, F-measure, and AUC three of the most popular performance metrics in this domain.

2.4 Summary

A wide range of metrics and classification methods have been incorporated with the Halstead metric suite for Software Fault Prediction (SFP), according to a thorough evaluation of previous research. The modeling process highlights the significance of a multifaceted approach to fault prediction by incorporating metrics such as Halstead, McCabe, Lines of Code (LoC), and Branch count as independent variables (IV). This underscores the importance of leveraging a comprehensive set of software metrics to improve the accuracy and reliability of predictive models. This research makes significant strides in advancing Software Fault Prediction (SFP) by integrating diverse metrics and state-of-the-art machine learning techniques. The study emphasizes a holistic approach, combining theoretical insights with empirical findings to propose robust models that enhance the accuracy, reliability, and applicability of fault prediction in complex software systems. Key findings of this review include:

The feature (software metrics) decomposition is not evaluated through any angle in SFP. That's show the gap in litterateur to evaluate the effectiveness of feature decomposition to improve ML models predictiveness in SFP.

Conducted a thorough evaluation of previous research to incorporate a wide range of metrics and classification methods with the Halstead metric suite for SFP.

Highlighted the importance of a multifaceted approach to fault prediction by integrating metrics such as Halstead, McCabe, Lines of Code (LoC), and Branch count as independent variables.

Demonstrated how diverse metrics like McCabe's Cyclomatic Complexity and Halstead metrics provide nuanced insights into software complexity and potential fault areas.

Improved fault detection capabilities and debugging efficiency by combining these metrics with advanced machine learning algorithms (e.g., decision trees, support vector machines, and neural networks).

Emphasized the need for continuous validation and empirical refinement of SFP models to enhance their reliability and generalization across software domains.

Showcased the use of diverse classification algorithms, including Logistic Regression (LR), Multi-Layer Perceptron (MLP), Random Forest (RF), Naïve Bayes (NB), and Least Squares Support Vector Machines (LSSVM) with different kernel functions.

Adopted a comprehensive performance evaluation strategy using performance measure such as AUC, accuracy and F-measure for robust model assessment.

Synthesized findings to propose a framework integrating the decomposed Halstead base metric with complementary software metrics and advanced classification algorithms to develop reliable SFP models.

Contributed to actionable insights for managing and mitigating software faults in complex software development environments.

Chapter 3

Methodology

Our objective is to evaluate the impact of the Decomposed Halstead base metric in SFP. To achieve this goal, we have designed a methodology as depicted in Figure 3.1, which would steer the execution of our experiment, elaborated in the next section. In the proposed methodology, our initial step involves the selection of case studies. The optimal choice for a case study would encompass a publicly available dataset along with accompanying source code. These case studies will serve as the foundation for the development of three distinct datasets. The first dataset, denoted as "Dataset-1", will encompass the Halstead metric suite as well as frequently reported valuable metrics utilized in SFP, such as Lines of Code (LoC) and McCabe. The second dataset, referred to as "Dataset-2", will consist of the same software metrics employed in "Dataset-1", with the exception of the Halstead base metrics. To obtain the Decomposed Halstead base metrics, the source code of the selected case studies will be parsed using a metrics extractor. The parsed Decomposed Halstead base metric will then be merged with "Dataset-2," resulting in the creation of a new dataset called "Dataset-3". This new dataset, "Dataset-3," will encompass both the Decomposed Halstead base metric and the SFP metrics selected in "Dataset-2". Subsequently, a machine learning algorithm will be employed to model the relationship between the independent variable (IV) and the dependent variable (DV) in both "Dataset-1" and "Dataset-3". Finally, the performance of "ML Model-1" and "ML Model-2" will be compared and analyzed utilizing various performance measures. In summary, the methodology comprises the following key phases:

3.pdf 3.pdf 3.pdf 3.pdf 3.pdf 3.pdf 3.pdf 3.pdf 3.pdf

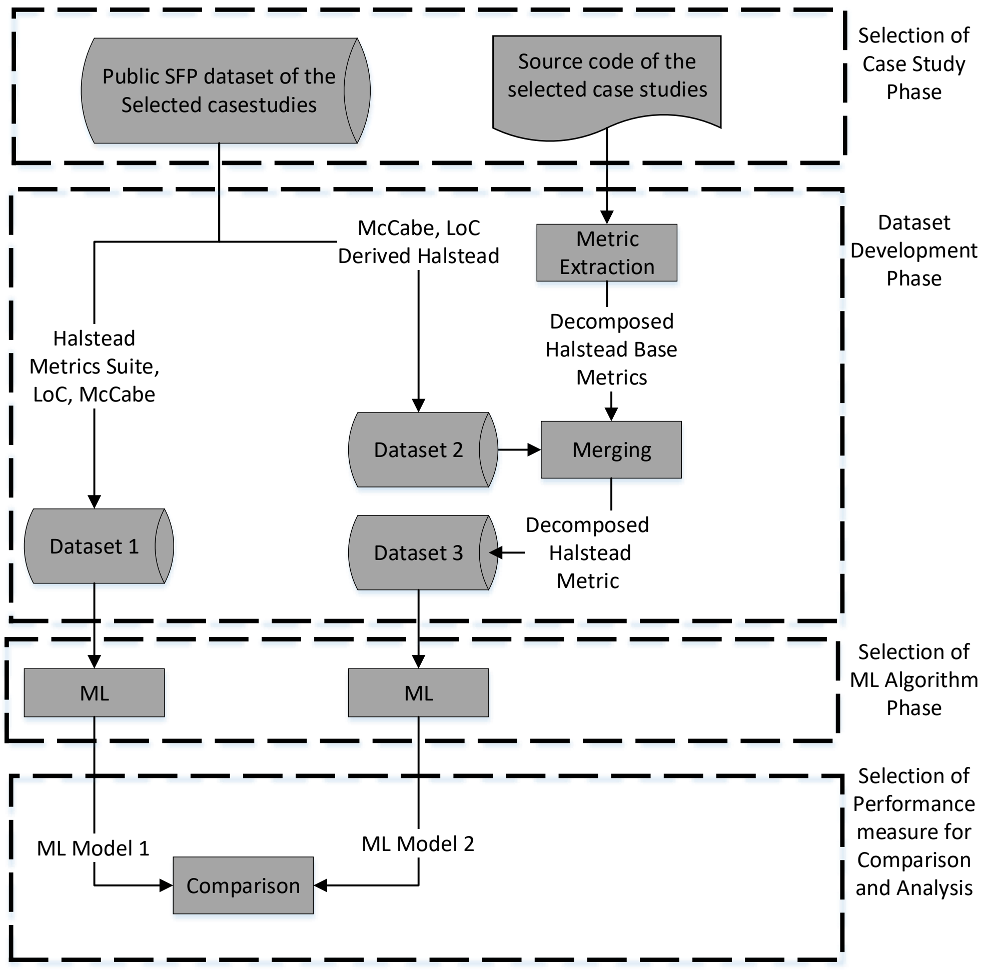
Public SFP dataset of the Selected casestudies
Source code of the selected case studies
Selection of Case Study Phase
McCabe, LoC Derived Halstead
Metric Extraction
Dataset Development Phase
Decomposed Halstead Base Metrics
Halstead Metrics Suite, LoC, McCabe
Dataset 2
Merging
Dataset 1
Dataset 3
Decomposed Halstead Metric
ML
ML
Selection of ML Algorithm Phase
Selection of Performance measure for Comparison and Analysis
ML Model 1
ML Model 2
Comparison

Figure 3.1: Flow Diagram For Proposed Approach

3.1 Selection of Case Studies

In this phase, a selection of case studies would be made, upon which subsequent processing shall be performed. It is widely acknowledged that ML-based empirical studies exhibit a high degree of bias due to the quality of data. This is largely attributed to the inadequacy of data and the absence of systematic data collection procedures. It is noteworthy that SFP has been executed using a diverse range of datasets, which may be classified into four categories based on their availability, namely: Private, Partially private, Partially public, and Public, as per [60]. In private datasets, neither the source code nor the fault information is provided, rendering studies based on these datasets non-repeatable. Partially private datasets offer access only to source code and/or metrics values, without fault information.

TABLE 3.1: Types of datasets w.r.t. availability of metrics values, fault information, and source code

Type of dataset	Metrics' values	Fault information	Source code
Private	✗	✗	✗
Partially private	✓	✓	✗
Partially public	✗	✓	✓
Public	✓	✓	✓

Partially public datasets typically provide access to both source code and fault data, but not metrics values, which must be extracted from the source code and mapped to fault data from the repository [60]. Public datasets, on the other hand, refer to datasets in which metrics values, source code, and fault data are publicly available for all modules in a software system. Table 3.1 illustrates these five dataset types. Since we aim to evaluate the performance disparity between Halstead base metrics and Decomposed Halstead base metrics, a suitable dataset for this task would be one that is at least partially publicly accessible.

3.2 Datasets Enhancement

After the identification of the case studies during the phase of case study selection, the subsequent task is dataset development. As previously discussed, the appropriate case studies for our experiment should possess publicly available software metrics datasets and their corresponding source code. To fulfill this requirement, we proceed to construct Dataset-1 using the chosen public dataset, specifically selecting the Halstead metric suite along with McCabe and LoC metrics. For the development of Dataset-2, we select all SFP metrics from the public dataset, which were previously included in dataset-1, excluding the Halstead base metrics. There is no publicly available dataset containing information on decomposed Halstead base metrics. Therefore, we will employ a metrics extractor on the source code of selected case studies to obtain these metrics. Existing metrics extractors have been examined to calculate the decomposed Halstead base metrics. However, it has been noted that these existing extractors possess three primary limitations:

1. The extractors have a lack of extensibility and, hence, may not be used to integrate with existing frameworks/extractors.
2. The extractors are metrics-specific and may not extract new metrics. Hence they are not easy to adapt to other metrics.
3. The extractors have an ambiguous interpretation of some metrics. Hence, more than one variant of the same metric exists which is reported in [83].

Taking into consideration the constraints and requirements of our experimentation, it is imperative to undertake the development of a custom-built extractor that possesses the capability to extract Halstead base metrics from the given source code. The primary function of our extractor involves parsing the source code of the designated case studies with the objective of extracting decomposed Halstead base metrics. Throughout the process of parsing the Decomposed Halstead base metrics suite, a hierarchical tree, as illustrated in Figure 4.1, will be employed. An algorithm 1 in the section 4.5, Where program statements will first be split into tokens, and then classified into operands and operators. The process will be sequential, with operands containing variables and constants being considered first, followed by operators. Within the operators, assignment, arithmetic, logical, and relational operators will be identified, while the remaining operators will be categorized as "others". Once the decomposed Halstead base metrics have been extracted, they are to be merged with Dataset-2 in a formal manner. During the parsing phase, it is imperative to preserve the information pertaining to the "Complete path of the source code file" as well as the "Name of Class" contained within that file. This information plays a crucial role in distinguishing between similar class names across multiple files and different classes within the same file. Dataset-2 also encompasses this essential information. The merging process culminates in the creation of Dataset-3, which encompasses the parsed Decomposed Halstead base metrics and the specifically chosen SFP metrics from Dataset-2. Subsequently, Dataset-1 and Dataset-3 will serve as inputs to the machine learning algorithm for the purposes of model construction and the execution of SFP.

3.3 Selection of ML Algorithm

Once the datasets have been developed, it's important to determine the suitable ML algorithm for modeling. Selecting the appropriate machine learning algorithms for software fault prediction involves considering various aspects to ensure optimal performance and effectiveness. Some common aspects are:

3.3.1 Nature of Data

Understand the nature and characteristics of the software development data available, including its volume, variety, velocity, and veracity. Determine whether the data is structured (e.g., code metrics, historical defect data) or unstructured (e.g., natural language bug reports). Choose machine learning algorithms that are well-suited to handle the specific data types and formats present in the software development environment.

3.3.2 Complexity of Patterns

Assess the complexity of patterns and relationships within the software development data. Select machine learning algorithms capable of capturing intricate patterns and non-linear relationships if the data exhibits high complexity. Consider the trade-offs between model complexity and interpretability, especially in domains where explain ability is crucial, such as software fault prediction.

3.3.3 Performance Requirements

Define performance metrics and objectives for the software fault prediction task, such as accuracy, precision, recall, F1-score, or area under the ROC curve (AUC). Choose machine learning algorithms that align with the desired performance metrics and can meet the specified performance requirements within the given constraints.

3.3.4 Scalability and Efficiency

Consider the scalability and computational efficiency of machine learning algorithms, particularly when dealing with large-scale software development datasets. Evaluate the algorithm's training and inference times to ensure practical feasibility within the software development lifecycle.

3.3.5 Interpretability and Explainability

Assess the interpretability and explainability of machine learning models, especially in safety-critical domains like software engineering. Choose algorithms that produce transparent and interpretable results to facilitate understanding and trust among stakeholders, such as developers and project managers.

3.3.6 Ensemble Methods

Explore ensemble learning techniques, such as random forests, gradient boosting, or stacking, to combine multiple machine learning models for improved predictive performance. Ensemble methods can mitigate the weaknesses of individual algorithms and enhance overall fault prediction accuracy and robustness.

3.3.7 Domain Expertise

Incorporate domain expertise and knowledge of software development practices into the algorithm selection process. Collaborate with software engineers, quality assurance professionals, and domain experts to identify relevant features, define appropriate labels, and validate the effectiveness of machine learning models.

3.3.8 Model Maintenance and Adaptation

Consider the ongoing maintenance and adaptation of machine learning models as software development practices evolve over time. Choose algorithms that support incremental learning or online learning to continuously update the model with new data and adapt to changing patterns and trends in the software development environment.

By carefully considering these aspects, software development teams can make informed decisions when selecting machine learning algorithms for software fault prediction, ultimately improving the effectiveness and reliability of fault prediction systems.

3.4 Selection of performance measures

The results of ML models are assessed by some performance measures. In classification, commonly refereed performance measures in our literature review are Accuracy, AUC, and F-measure. Selecting appropriate performance measures for machine learning algorithms in software fault prediction is crucial for evaluating their effectiveness and guiding decision-making processes. Several aspects should be considered when choosing performance measures.

3.4.1 Nature of the Problem

Determine whether the software fault prediction problem is binary (fault vs. no fault), multi-class (predicting fault severity levels), or regression-based (predicting the number of faults). Select performance measures that are suitable for the specific problem type. For binary classification, metrics like accuracy, precision, recall, F1-score, and ROC-AUC are commonly used. For regression tasks, metrics such as mean absolute error (MAE) and mean squared error (MSE) are appropriate.

3.4.2 Imbalance in the Dataset

Assess the balance between fault and non-fault instances in the dataset. Imbalanced datasets, where one class is significantly more prevalent than the other, can skew performance measures. Use metrics like precision, recall, F1-score, and area under the precision-recall curve (PR AUC) that are robust to class imbalance.

3.4.3 Cost Sensitivity

Consider the costs associated with false positives (misclassifying a non-fault as a fault) and false negatives (misclassifying a fault as a non-fault). Utilize performance measures that account for these costs, such as cost-sensitive versions of precision, recall, and F1-score, or cost curves that plot performance against different misclassification costs.

3.4.4 Interpretability

Choose performance measures that are easily interpretable and provide actionable insights for software development teams. Metrics like accuracy, precision, and recall offer straightforward interpretations and are widely understood in the software engineering domain.

3.4.5 Threshold Selection

Determine the threshold for classifying instances as fault or non-fault based on the predicted probabilities or scores provided by the machine learning model. Consider using performance measures that assess model performance across different threshold values, such as precision-recall curves and ROC curves.

3.4.6 Domain Specific Requirements

Take into account any specific requirements or constraints of the software development environment. For example, in safety-critical systems, minimizing false negatives (missing faults) may be prioritized over false positives (false alarms), leading to a focus on metrics like recall.

3.4.7 Validation and Cross Validation

Validate the performance of machine learning algorithms using appropriate validation techniques such as holdout validation, k-fold cross-validation, or bootstrapping. Use consistent performance measures across different validation folds to ensure robust and reliable evaluation results.

3.4.8 Comparative Analysis

Conduct a comparative analysis of multiple machine learning algorithms to identify the most suitable approach for software fault prediction. Use consistent performance measures for all algorithms to facilitate fair comparisons and informed decision-making.

By considering these aspects and selecting appropriate performance measures, software development teams can effectively evaluate the performance of machine learning algorithms in software fault prediction and make informed decisions to improve software quality and reliability.

3.5 Summary

This chapter begins with the selection of case studies and the development of datasets to ensure comprehensive and compatible data for analysis. Various machine learning algorithms, such as Logistic Regression, Decision Trees, Random Forests, and Support Vector Machines, are employed to model fault prediction. The performance of these models is assessed using standard metrics, including Accuracy, F-measure, and AUC. This systematic approach provides a robust framework for analyzing the impact of decomposed Halstead base metrics in enhancing predictive capabilities.

Chapter 4

Decomposition of Halstead Metric Suite

To improve accuracy, and interoperability in software analysis, complicated aggregated metrics are decomposed into their basic components through the process of decomposition. This approach offers a more detailed knowledge of the features affecting software quality and fault prediction by decomposition of aggregated complex feature. By identifying certain areas of complexity and possible flaws, this better understanding helps developers more efficiently address the underlying causes of problems and enhances fault detection accuracy, maintainability, and customization. In order to take advantage of decomposition for fault prediction, the conventional Halstead measurements are decomposed into more detailed metrics in this dissertation. The Halstead Software Metrics were chosen for this study due to their relevance in evaluating software complexity and their alignment with the objectives of the research. The decision to focus on Halstead, instead of other metrics, is justified based on the following factors.

Granular complexity measurement: By measuring operators, operands, and their occurrences, Halstead offers a thorough analysis of code complexity. This level of detail is necessary to comprehend how various code elements affect fault proneness.

Feature compatibility for ML models: Different kinds of operators and operands, which are the decomposed components of Halstead, are significant features that enhance the predictive precision of machine learning models.

Proven effectiveness in fault prediction: Prior research has validated the efficacy of Halstead metrics suite in fault-prone module identification. Their established use in empirical studies supports their inclusion as a robust and reliable set of features for this research.

Limitations of Alternative Metrics:

McCabe's Cyclomatic Complexity: Focuses solely on control flow and does not provide insight into the detailed usage of operators and operands.

Lines of Code (LoC): Although simple, LoC lacks the granularity required to identify specific fault-prone areas and often correlates with larger modules rather than complexity itself.

By selecting Halstead, this study aims to leverage detailed software metrics that capture the nuanced contributions of code components to fault-proneness, ultimately leading to a more accurate and interpretable fault prediction model.

4.1 Halstead's Metrics: an Overview

Halstead's (1977) software science was one of the first attempts to provide a theory of software measurement that is oriented toward program code [84–87]. Keeping in view the elaboration of Halstead, every computer software program is a collection of numerous tokens that can safely be categorized into two classes. The first one is operators and the second is operands. Based upon these tokens, Halstead has derived four base metrics and numerous derived metrics [88]. Derived metric can be computed by frequency of tokens. [84].

4.1.1 Halstead Base Metrics

Halstead Base Metrics are fundamental components of the Halstead Complexity Measures, which are used to quantify the complexity of software.

These metrics were introduced by Maurice H. Halstead in 1977 as part of his Software Science theory.

They are derived from the source code of a program and focus on the number of operators and operands in the code. The description of base metrics are as in Table 4.1

n_1 **(Number of distinct operators):** This represents the unique operators used in the program. Operators include arithmetic symbols, relational operators, logical operators, and others depending on the programming language.

n_2 **(Number of distinct operands):** This represents the unique operands used in the program. Operands are variables, constants, or values that operators manipulate.

N_1 **(Total number of operators):** This counts how many times operators are used throughout the code.

N_2 **(Total number of operands):** This counts how many times operands are used in the code.

Potential operators: refer to the minimal or ideal set of operators necessary to implement a specific functionality optimally. They represent the simplest logical structure of a program.

Using potential operators helps in evaluating how efficiently a program is written by comparing actual operators to the minimum possible operators for the same solution.

Table 4.1: Halstead Base Metrics

Metrics	Description
n_1:	Total count of unique operators
n_2:	Total count of unique operands
N_1:	Total count of all operators
N_2:	Total count of all operands
n_1*:	Total count of minimum operators.
n_2*:	Total count of minimum operands.

All of the metrics proposed by Halsteads are also named as Software Science metrics.

4.1.2 Halstead Derived Metrics

The Halstead base metrics are used to compute numerous derived metrics. This section briefly describes the driven metrics proposed by Halstead.

The length of a program P is donated by N, can mathematically be written as:

$$N = N_1 + N_2 \tag{4.1}$$

The vocabulary of a program P is donated by n, can mathematically be written as::

$$n = n_1 + n_2 \tag{4.2}$$

Program volume V of a program P is donated by V, can mathematically be written as:

$$V = N * log_2 n \tag{4.3}$$

Program potential (minimal) volume, which is denoted by V* can mathematically be written as:

$$V^* = (2 + n_2^*) log_2 (2 + n_2^*) \tag{4.4}$$

Program level of a program P with volume V can mathematically be computed as:

$$L = \frac{V^*}{V} \tag{4.5}$$

The maximum value for L is 1.

Program difficulty of a program P is donated by D, can mathematically be written as:

$$D = \frac{1}{L} \tag{4.6}$$

The program level estimator of a program P is donated by ($\hat{L}$), can mathematically be written as:

$$\hat{L} = 2 * (n_2)/(n_1)(N_2) \tag{4.7}$$

The intelligent content of a program P is a denoted by I, can mathematically be written as:

$$I = \frac{\hat{L}}{V} \tag{4.8}$$

Programming effort of a program P is denoted by E, can mathematically be written as:

$$E = \frac{V}{L} = \frac{n_1 n_2 log_2 n}{2N_2} \tag{4.9}$$

The required programming time of a program P is denoted by R, and can mathematically be written as:

$$T = \frac{E}{S} = \frac{n_1 n_2 log_2 n}{2N_2 S} \tag{4.10}$$

where S is the Stroud number which is set to 18 by the software scientists.

4.1.3 Miscellaneous Halstead Metrics Suite Extension

Numerous studies propose various extensions to the Halstead metrics suite, like Halstead Number of Derived Bugs (HNDB) [89], minimal operator count [90]. However, this dissertation document focused on the basic Halstead metric suite. The reason is its wide acceptability and line-level coverage. The Halstead metric suite comprises four base metrics and ten derived metrics which are derived out of the base metrics. The four base metrics are Total operators N_1, Total operands N_2, Unique operators n_1, and Unique operands n_2. Likewise, these base metrics are composed of operators and operands. According to the definition of Halstead, the software program is a composition of tokens. Each token can either be an operator or an operand. Operand includes variables and constants. While all other tokens are included in operators [85] which can safely be extracted from various languages [91]. The following subsection would elaborate the decomposition of Halstead operators and Halstead operands.

4.2 Decomposition of Halstead Operators

The definition of Halstead operator is more general than that of the operators defined in conventional programming languages like C, C++, Java, etc. [92–96]. For example, brackets, function name, semicolon, colon, punctuation marks, etc. are Halstead operators but these are not considered as operators in the C, C++, Java etc. Moreover, there are few operators that are present in some languages while missing in some others. Like, increment/decrement operators are present in C, C++, Java, PhP and they are not present in Python language. Keeping in view this difference, we conclude that few Halstead operators are considered as operators by all major languages, while few Halstead operators are either do not defined as operators by all languages or by few languages. This understanding lead us to answer our RQ1, possible five broad categories that are applicable to all major programming languages.

Assignment operators, that are declared as Assignment operators in the conventional languages. The assignment may be performed explicitly by using $=$ operator or in combination with other operators like, $+=$, $-=$ or sometime implicitly like, $--$, $++$.

Arithmetic operators, that are declared as Arithmetic operators in the conventional languages. Like assignment operators, arithmetic operators may be declared explicitly by using $+, -$, etc operator or in combination with other operators like $+=, -=$.

Logical operators that are declared as Logical operators in the conventional languages like && (AND), || (OR).

Relational operators that are declared as Relational operators in the conventional languages like $\leq$, $\geq$.

Others constitutes the rest of Halstead operators like brackets and function names are:

For instance, if A is base metric, it will be decomposed into $a_1, a_2, ..., a_n$ such that the value of metric A is equal to the sum of all the values of decomposed metrics $a_1, a_2, ..., a_n$. In this dissertation, we demonstrate the effectiveness of decomposed Halstead base metric using Java language.

The operators that are declared in Java can be placed into the five categories discussed earlier. The table 4.2 illustrates the Java operators with their corresponding categories. $=$ belongs to assignment operator category.

There are certain operators which perform both arithmetic and assignment operations These operators include, +=, -=, *=, /=, %=, &= ˆ=, |=, <<=, >>=, ++, –. Therefore these operators are placed in both categories +, -, *, / and % belongs to arithmetic operators category. &&, ||, ! belongs to logical operators category. ==, <, <=, >, >=, ! = belongs to relational operator category. Rest of the operators like, bitwise operators, brackets, etc. are all belongs to Others category.

Table 4.2: Java operators and their category

Category	**Operator**	**Description**
Assignment	$=$	Simple assignment operator
	$+=$	Addition and assignment operator
	$-=$	Subtraction and assignment operator
	$*=$	Multiplication and assignment operator
	$/=$	Divide and assignment operator

	% =	Modulus and assignment operator
	& =	Bitwise and assignment operator
	ˆ=	Bitwise exclusive OR and assignment operator
	|=	Bitwise inclusive OR and assignment operator
	<<=	Left shift and assignment operator
	>>=	Right shift and assignment operator
Arithmetic	+	Additive operator
	−	Subtraction operator
	∗	Multiplication operator
	/	Division operator
	%	Remainder operator
	+	Unary plus operator
	−	Unary minus operator
	++	Increment operator
	−−	Decrement operator
Logical	&&	Conditional-AND
	| |	Conditional-OR
	!	Logical complement operator
Relational	==	Equal to
	>	Greater than
	! =	Not equal to
	>=	Greater than or equal to
	<	Less than
	<=	Less than or equal to
Others	Function name	Any function name in a program
	Class name	Any class name in a program
	{, }, [,]	Brackets
	int, float, double, etc.	Data types
	; ,	Special symbols

The above categorization has been used in case studies (Section 5.1) for empirical evaluation of decomposed Halstead base metrics.

The Halstead operators along with corresponding decomposed metrics are shown in Table 4.3.

Table 4.3: Halstead operators with their corresponding decomposed operators

Base metrics	Decomposed metrics	Description
Total operators (N_1)	As	Total assignment operators
	A	Total arithmetic operators
	R	Total relational operators
	Log	Total logical operators
	O	Total operators other than assignment, arithmetic, relational, and logical operators
Unique operators (n_1)	as	Unique assignment operators
	a	Unique arithmetic operators
	r	Unique relational operators
	log	Unique logical operators
	o	Unique operators other than assignment, arithmetic, relational, and logical operators

4.3 Decomposition of Halstead Operands

In conventional computer programming, an operand is a term used to describe any object that is capable of being manipulated. In Halstead's manuscript operands are composed of two mutually exclusive types, i.e., variables and constants. A variable is a data item whose value can be changed during the program's execution. A constant is a literal for representing a fixed value in source code. For instance, in the following code snippet of Java language.

```
1.  int x = 5;
2.  final int y=10;
3.  char z='b';
```

x, y, and z are variables, while 5, 10 and 'b' are constants, rest of the tokens i.e., int, final, char, =, and ;(semicolon) are all operators. Keeping in view such decomposition, if A is base metric, it will be decomposed into $a_1, a_2, ..., a_n$ such that the value of metric A is equal to the sum of all the values of decomposed metrics $a_1, a_2, ..., a_n$. The Halstead operands along with corresponding decomposed metrics are shown in Table 4.4.

Table 4.4: Halstead operands with their corresponding decomposed operands

Base metrics	Decomposed metrics	Description
Total operands (N_2)	Var	Total variables
	C	Total constants
Unique operands (n_2)	var	Unique constants
	c	Unique variables

The decomposed forms of operators and operands apply to all major programming languages, such as C, C++, and Java. A comparable decomposition structure is used in all of these languages, which aids in the comprehension and classification of the components involved in code execution. A clearer and more structured depiction of the operators' and operands' functions and interactions is made possible by this decomposition, which divides the operators and operands into mutually exclusive classes. This hierarchical classification is shown in Figure 4.1, where each class corresponds to a different group of operands or operators. For example, operands can be classified as variables and constants, and operators can be classified as Assignment, arithmetic, logical, and relational, among others. Better understanding and analysis of programming structures are made possible by this structured approach, which also makes it simpler to teach, learn, and apply these principles in a variety of programming contexts.

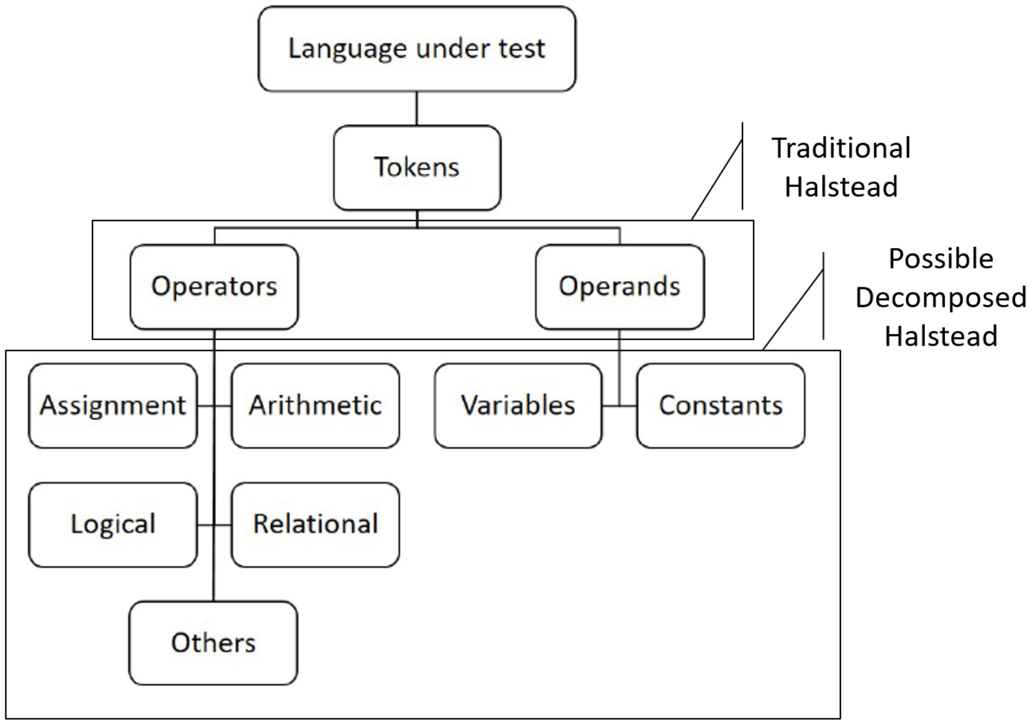

FIGURE 4.1: Hierarchy of decomposed operators and operands

The universal classification scheme depicted in the figure 4.1 highlights the ways that various programming languages handle operators and operands similarly. Programmers can become more versatile and proficient in software development by better translating their knowledge and skills across many languages by comprehending this hierarchy.

4.4 Mathematical Formulation of Decomposed Halstead Operators and Operands

This section addresses the mathematical representation of decomposed Halstead base metrics to improve the complexity analysis of software modules. The summing and weighted contributions of decomposed operators and operands are defined by the following equations 4.11 to 4.18, which show how we can compute decomposed Halstead operator and operands to improve machine learning models predictive power.

The total count of operators N_1 must be decomposed as the total sum of decomposed operators like $\{N_{11}, N_{12}, \ldots, N_{1m}\}$ count is equal to total count of operators N_1 as shown in equation 4.11.

$$N_1 = \sum_{i=1}^{m} N_{1i} \tag{4.11}$$

The total count of unique operators n_1 must be decomposed as the total sum of decomposed unique operators like $\{n_{11}, n_{12}, \ldots, n_{1m}\}$ count is equal to total count of unique operators n_1 as shown in equation 4.12.

$$n_1 = \sum_{i=1}^{m} n_{1i} \tag{4.12}$$

The total count of operands N_2 must be decomposed as the total sum of decomposed operands like $\{N_{21}, N_{22}, \ldots, N_{2m}\}$ count is equal to total count of operands N_2 as shown in equation 4.13.

$$N_2 = \sum_{i=1}^{m} N_{2i} \tag{4.13}$$

The total count of unique operands n_2 must be decomposed as the total sum of decomposed unique operands like $\{n_{21}, n_{22}, \ldots, n_{2m}\}$ count is equal to total count of unique operands n_2 as shown in equation 4.14.

$$n_2 = \sum_{i=1}^{m} n_{2i} \tag{4.14}$$

The frequency of operators and operands in a particular software module is compiled in these summaries. In the context of decomposition, these numbers show the raw complexity of a program, while typical Halstead base metrics fail to capture precise interactions between types of separate operators and operands.

At level 2, decomposition of each decomposed total operator like N_{1i} is further decomposed such that that total sum of decomposed operator at level 2 like $\{N_{1i.1}, N_{1i.2}, \ldots, N_{1i.m}\}$ count is equal to the total count of N_{1i} as depicted in equation 4.15.

$$N_{1i} = \sum_{j=1}^{m} N_{1i.j} \tag{4.15}$$

The decomposition of each decomposed unique operator like n_{1i} is further decomposed such that that total sum of decomposed operator at level 2 like $\{n_{1i.1}, n_{1i.2}, \ldots, n_{1i.m}\}$ count is equal to the total count of n_{1i} as depicted in equation 4.16.

$$n_{1i} = \sum_{j=1}^{m} n_{1i.j} \tag{4.16}$$

The decomposition of each decomposed total operands like N_{2i} is further decomposed such that that total sum of decomposed operand at level 2 like $\{N_{2i.1}, N_{2i.2}, \ldots, N_{2i.m}\}$ count is equal to the total count of N_{2i} as depicted in equation 4.17.

$$N_{2i} = \sum_{j=1}^{m} N_{2i.j} \tag{4.17}$$

The decomposition of each decomposed unique operand like n_{2i} is further decomposed such that that total sum of decomposed operand at level 2 like $\{n_{2i.1}, n_{2i.2}, \ldots, n_{2i.m}\}$ count is equal to the total count of n_{2i} as depicted in equation 4.18.

$$n_{2i} = \sum_{j=1}^{m} n_{2i.j} \tag{4.18}$$

4.5 Algorithm to Compute Decomposed Halstead Base Metrics from the Code

The algorithm 1 is designed to decompose Halstead base metrics into finer subcategories of operators and operands to enhance machine learning efficiency. The input to the algorithm is the program source code $\mathcal{P}$, and the output is a set of decomposed metrics for operators and operands, including $N_1^{\text{As}}, N_1^{\text{A}}, N_1^{\text{R}}, N_1^{\text{Log}}, N_1^{\text{O}}$ (representing assignment, arithmetic, relational, logical, and other operators, respectively) and $N_2^{\text{Var}}, N_2^{\text{Const}}$ (representing variables and constants, respectively). The algorithm begins by initializing counters for

total operators (N_1) and operands (N_2) and their respective subcategories. In the first major step, the source code $\mathcal{P}$ is tokenized into a list of tokens $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$. This tokenization process allows for systematic processing of individual components of the source code, ensuring that each token can be categorized into its appropriate metric.

The decomposition process starts with the classification of operators. For each token $t_i \in \mathcal{T}$, the algorithm identifies whether it is an assignment operator (N_1^{As}), arithmetic operator (N_1^{A}), relational operator (N_1^{R}), logical operator (N_1^{Log}), or falls into the "other" operator category (N_1^{O}). Each operator is incrementally added to its respective subcategory, while the total operator count (N_1) is updated. Similarly, operands are categorized into variables (N_2^{Var}) and constants (N_2^{Const}) based on their characteristics. Each occurrence of an operand is recorded in its subcategory, and the total operand count (N_2) is updated. Once all tokens have been processed, the total metrics are computed: N1 $= N_1^{\text{As}} + N_1^{\text{A}} + N_1^{\text{R}} + N_1^{\text{Log}} + N_1^{\text{O}}$ and N2 $= N_2^{\text{Var}} + N_2^{\text{Const}}$. The final results, including the decomposed operators and operands, are outputted for analysis, allowing for fine-grained evaluation of the source code's complexity. This decomposition is critical for optimizing machine learning algorithms by providing detailed metrics that enhance model training and performance.

Algorithm 1 Decomposing Halstead Base Metrics for Machine Learning Efficiency

```
1: Input: Program source code $\mathcal{P}$
2: Output: Decomposed operators $\{N_1^{\text{As}}, N_1^{\text{A}}, N_1^{\text{R}}, N_1^{\text{Log}}, N_1^{\text{O}}\}$ and operands $\{N_2^{\text{Var}}, N_2^{\text{Const}}\}$
3: Initialize $N_1 \leftarrow 0$, $N_2 \leftarrow 0$    ▷ Counters for total operators and operands
4: Initialize $N_1^{\text{As}} \leftarrow 0$, $N_1^{\text{A}} \leftarrow 0$, $N_1^{\text{R}} \leftarrow 0$, $N_1^{\text{Log}} \leftarrow 0$, $N_1^{\text{O}} \leftarrow 0$    ▷ Operator subcategories
5: Initialize $N_2^{\text{Var}} \leftarrow 0$, $N_2^{\text{Const}} \leftarrow 0$    ▷ Operand subcategories
   Step 1: Tokenize the source code
6: Parse $\mathcal{P}$ into a list of tokens $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$
   Step 2: Decompose operators
7: for each token $t_i \in \mathcal{T}$ do
8:     if $t_i$ is an assignment operator (=, +=, -=, etc.) then
9:         $N_1^{\text{As}} \leftarrow N_1^{\text{As}} + 1$    ▷ Count assignment operators
10:    else if $t_i$ is an arithmetic operator (+, -, *, /, %) then
11:        $N_1^{\text{A}} \leftarrow N_1^{\text{A}} + 1$    ▷ Count arithmetic operators
12:    else if $t_i$ is a relational operator (==, !=, >, <, >=, <=) then
13:        $N_1^{\text{R}} \leftarrow N_1^{\text{R}} + 1$    ▷ Count relational operators
14:    else if $t_i$ is a logical operator (&&, ||, !) then
15:        $N_1^{\text{Log}} \leftarrow N_1^{\text{Log}} + 1$    ▷ Count logical operators
16:    else
17:        $N_1^{\text{O}} \leftarrow N_1^{\text{O}} + 1$    ▷ Count other operators
18:    end if
19:    $N_1 \leftarrow N_1 + 1$    ▷ Increment total operator count
20: end for
    Step 3: Decompose operands
21: for each token $t_i \in \mathcal{T}$ do
22:    if $t_i$ is a variable name then
23:        $N_2^{\text{Var}} \leftarrow N_2^{\text{Var}} + 1$    ▷ Count variable operands
24:    else if $t_i$ is a constant (e.g., number, string, etc.) then
25:        $N_2^{\text{Const}} \leftarrow N_2^{\text{Const}} + 1$    ▷ Count constant operands
26:    end if
27:    $N_2 \leftarrow N_2 + 1$    ▷ Increment total operand count
28: end for
    Step 4: Compute summary statistics
29: Total Operators (N1) $\leftarrow N_1^{\text{As}} + N_1^{\text{A}} + N_1^{\text{R}} + N_1^{\text{Log}} + N_1^{\text{O}}$
30: Total Operands (N2) $\leftarrow N_2^{\text{Var}} + N_2^{\text{Const}}$
    Step 5: Output results
31: Return the decomposed operators:
32: $\{N_1^{\text{As}}, N_1^{\text{A}}, N_1^{\text{R}}, N_1^{\text{Log}}, N_1^{\text{O}}\}$
33: Return the decomposed operands:
34: $\{N_2^{\text{Var}}, N_2^{\text{Const}}\}$
35: Return total metrics:
36: N1 $= N_1$, N2 $= N_2$
```

4.6 Justification for Halstead Base Metrics Decomposition

In order to overcome the hidden information may be associated to fault proneness of aggregated software metrics which could mask the influence of individual components on software complexity and fault prediction. like decomposition of Halstead base metrics.

The necessity and advantages of this decomposition method are explained in detail in this section.

4.6.1 Necessity of Decomposition

Traditional Halstead base metrics, while useful for estimating overall program complexity, often fail to capture the nuanced contributions of specific operators and operands for fault prediction.

For instance, different operators like control flow operators (if, for, while), arithmetic operators (+, -, *) etc may have a more significant impact on fault prediction compared to aggregated metric or Without decomposition.

4.6.2 Benefits of Decomposition

The decomposition of Halstead base metrics provides the following benefits.

4.6.2.1 Granular Feature Representation

A more comprehensive features set is made possible by the decomposition of operators and operands, which enhances the model's ability to identify pertinent patterns in the data.

4.6.2.2 Enhanced Model Interoperability

Decomposed metrics analysis makes it feasible to pinpoint particular kinds of operators or operands that have a strong correlation with errors, offering useful information for enhancing software quality.

4.6.2.3 Improved Prediction Performance

The validity of this strategy is supported by empirical findings showing that machine learning models trained on decomposed features have better classification accuracy and robustness.

This decomposition framework improves the models' interoperability and prediction performance, leading to a more thorough comprehension of fault-prone modules.

4.7 Summary

The chapter explores the decomposition of Halstead base metrics into more granular components to enhance their utility in software complexity analysis and machine learning applications. Halstead's original framework categorizes program tokens into operators and operands, deriving four base metrics and various derived metrics to quantify software complexity. This work breaks down operators into categories such as assignment, arithmetic, logical, relational, and miscellaneous, while operands are divided into constants and variables. By creating a hierarchical classification applicable across programming languages, this decomposition provides a more detailed and standardized analysis, improving both program comprehension and empirical evaluations in case studies, particularly using Java.

Chapter 5

Experimentation and Results

Keeping in view the methodology used in the articles which are discussed in the literature, our experiment comprises four components, i.e., Case study, Set of variables, Modeling algorithm, and performance measure used.

The datasets used in the literature does not contain information about the decomposed Halstead base metrics (Chapter 4 elaborates). Therefore, we build our own datasets. However, we need to have the source code and bug information of the projects. Consequently, we take five software projects (Section 5.1 elaborates) that have both the source code and fault information.

The assessment of the decomposed Halstead metrics has been accomplished by making four experiments having following three set of variables respectively, i.e.,
Set 1: {Halstead base metrics}
Set 2: {Halstead derived metrics, McCabe, LoC }
Set 3: {Decomposed Halstead Base Metrics}

The literature review indicates the frequently used ML algorithms which are elaborated in Section 5.4.

The evaluation of the results has been carried out by using Accuracy, F-measure, and AUC. These performance measures are elaborated in Section 5.6.

The graphical representation of our experiments are shown in Figure 5.1.

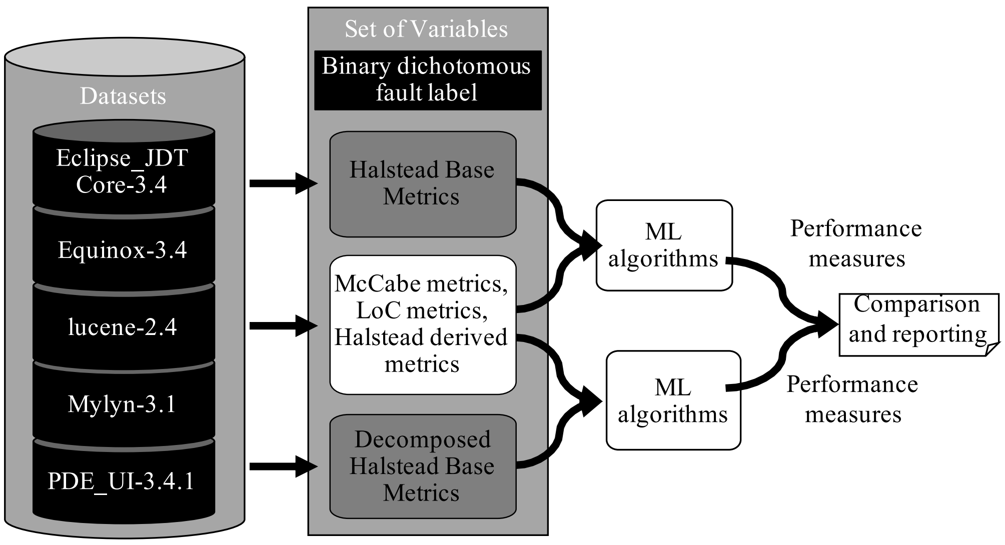

FIGURE 5.1: Experimental design

5.1 Case Study

For the experimentation purpose we selected following five datasets for being their public source code along with fault information.

1. Apache Lucene 2.4[1]
2. Eclipse equinox framework 3.4[2]
3. Eclipse JDT Core 3.4[3]
4. Eclipse PDE UI 3.4.1[4]
5. Mylyn 3.1[5]

These object oriented based projects are developed in Java and are publically available. Toth *et al.* assigned fault labels using bug tracking system [97].

Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that

[1]lucene.apache.org
[2]www.eclipse.org/equinox/
[3]www.eclipse.org/jdt/core/
[4]www.eclipse.org/pde/pde-ui/
[5]www.eclipse.org/mylyn/

requires full-text search, especially cross-platform. Apache Lucene is an open-source project available for free download.

Eclipse JDT Core is a Java infrastructure of the Java IDE. It includes an incremental Java compiler. In particular, it allows to run and debug code that still contains unresolved errors. It provides a Java-centric view of a project. It also carries a Java document model providing API for manipulating a structured Java source document.

Eclipse PDE UI provides a comprehensive set of tools to create, develop, test, debug and deploy Eclipse plug-ins. PDE UI also provides multi-page editors that centrally manage all manifest files of a plug-in or feature. It carries new project creation wizards to create a new plug-in, fragment, feature, feature patch, and update sites.

Eclipse equinox is an implementation of the OSGi core framework specification, a set of bundles that implement various optional OSGi services and other infrastructure for running OSGi-based systems. It is responsible for developing and delivering the OSGi framework implementation used for all of Eclipse. The Equinox OSGi core framework implementation is used as the reference implementation. The goal of the Equinox project is to be a first-class OSGi community and foster the vision of Eclipse as a landscape of bundles.

Mylyn is the task and application life cycle management framework for Eclipse. It provides a revolutionary task-focused interface and a task management tool for developers.

5.2 Selected Features

In our experimentation, the dependent variable is a binary dichotomous fault label, i.e., *fp* and *nfp*. Since the selected dataset contains the numerical fault label, we transformed it into binary using the following rulings shown in Equation 5.1.

$$Label = \begin{cases} fp & No.of\, faults > 0 \\ nfp & otherwise \end{cases} \tag{5.1}$$

The ratio of fault/fault-free is shown in Figure 5.2.

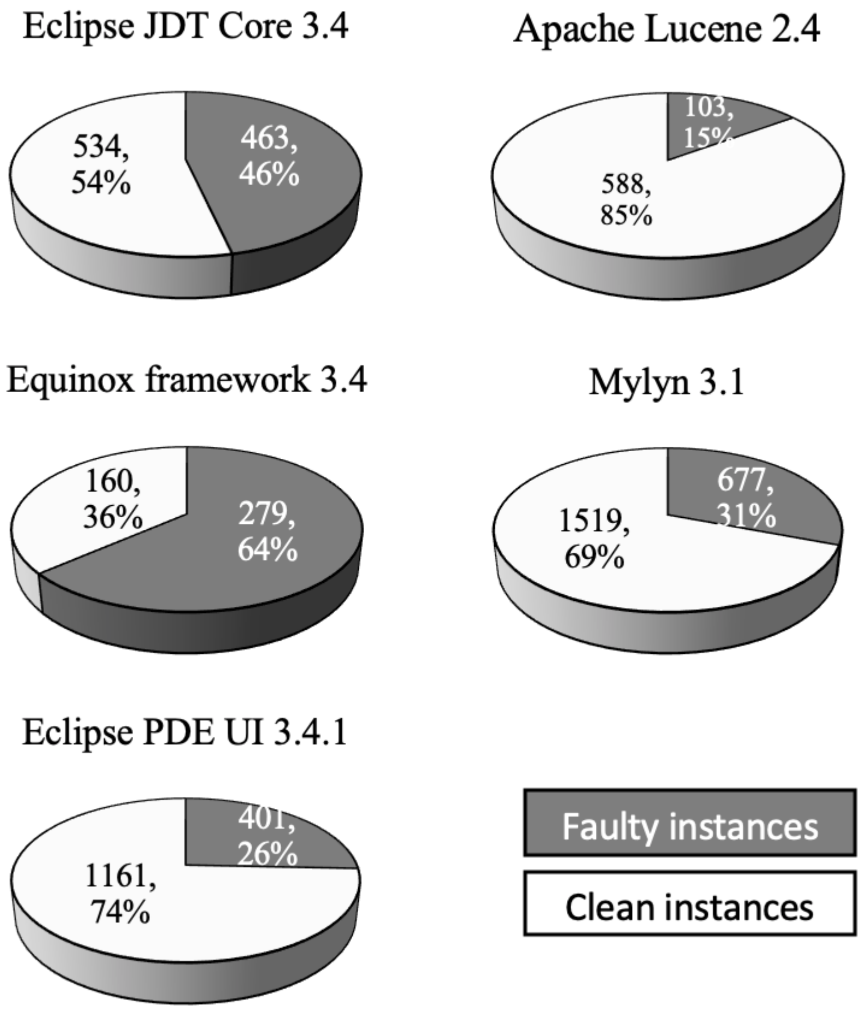

FIGURE 5.2: Ratio of fault and fault free in the five datasets

TABLE 5.1: Features in Set-1

Feature Short Form	Description
n_1	Total count of unique operators
n_2	Total count of unique operands
N_1	Total count of all operators
N_2	Total count of all operands

TABLE 5.2: Features in Set-2

Feature Short Form	Description
N	The Halstead program length.
n	The Halstead program vocabulary.
V	The Halstead program volume.
L	The Halstead program level.
D	The Halstead program difficulty.
I	The Halstead program intelligent.
E	The Halstead program effort.
T	The Halstead program time.
LoC	The total number of lines of the program.
LoEx	The total number of executable lines of the program.
LoB	The total number of blank lines of the program.
LoCm	The total number of comment lines of the program.
LoCoCm	The total number of code and comments lines of the program.
v(G)	The cyclomatic complexity of the program.
vi(G)	The design complexity of the program.
ev(G)	The essential complexity of the program.

TABLE 5.3: Features in Set-3

Feature Short Form	Description
As	Total assignment operators
A	Total arithmetic operators
R	Total relational operators
Log	Total logical operators
O	Total operators other than assignment, arithmetic, relational, and logical operators
Var	Total variables

C	Total constants
as	Unique assignment operators
a	Unique arithmetic operators
r	Unique relational operators
log	Unique logical operators
o	Unique operators other than assignment, arithmetic, relational, and logical operators
var	Unique constants
c	Unique variables

The independent variables comprise three feature sets, i.e., Halstead base metrics, Halstead derived metrics, LoC metric suite McCabe metric suites, and Decomposed Halstead base metrics. These feature set are placed in three distinct sets as follows:

Set 1: {Halstead base metrics} Set 2: {McCabe, LoC , Halstead derived metrics} Set 3: {Decomposed Halstead Base Metrics}

The first experiment comprises the features of Set-1 and Set-2, while the second, third, and fourth experiment comprises the features of Set-2 and Set-3. The detail description and their distribution in the experiments are shown in Table 5.1, 5.2, and 5.3. The total number of features at in each experiment through decomposition is depicted in table 5.4.

Table 5.4: Feature count in each experiments through decomposition

Experiments	McCab	LoC	Halstead D	Halstead B	Halstead DB	Total
Exp1 L0	3	5	7	4	0	19
Exp2 L1	3	5	7	2	10	27
Exp3 L1	3	5	7	0	14	29
Exp4 L2	3	5	7	0	54	69

5.3 Data Preprocessing

The data preprocessing comprises the following two steps:

1. Conversion of numerical fault label to binary fault label as shown in Equation 5.1.

2. Assessing the validation of data. In this activity we analyzed and ensure the absence of missing values, out of range values, Null value, invalid value (like negative in total operators), etc.

5.4 ML Modeling

The literature review section indicates the frequent usage and reported effectiveness of the following six ML algorithms. That provides a bases of the usage of these algorithms in our experiments also.

5.4.1 Logistic Regression

Logistic regression (as depicted in figure 5.3) model is a statistical model that models the probability of one event (out of two alternatives) taking place by having the log-odds (the logarithm of the odds) for the event be a linear combination of one or more independent variables (”predictors”). Formally, in binary logistic regression there is a single binary dependent variable, coded by an indicator variable, where the two values are labeled “0” and “1”, while the independent variables can continuous variable.

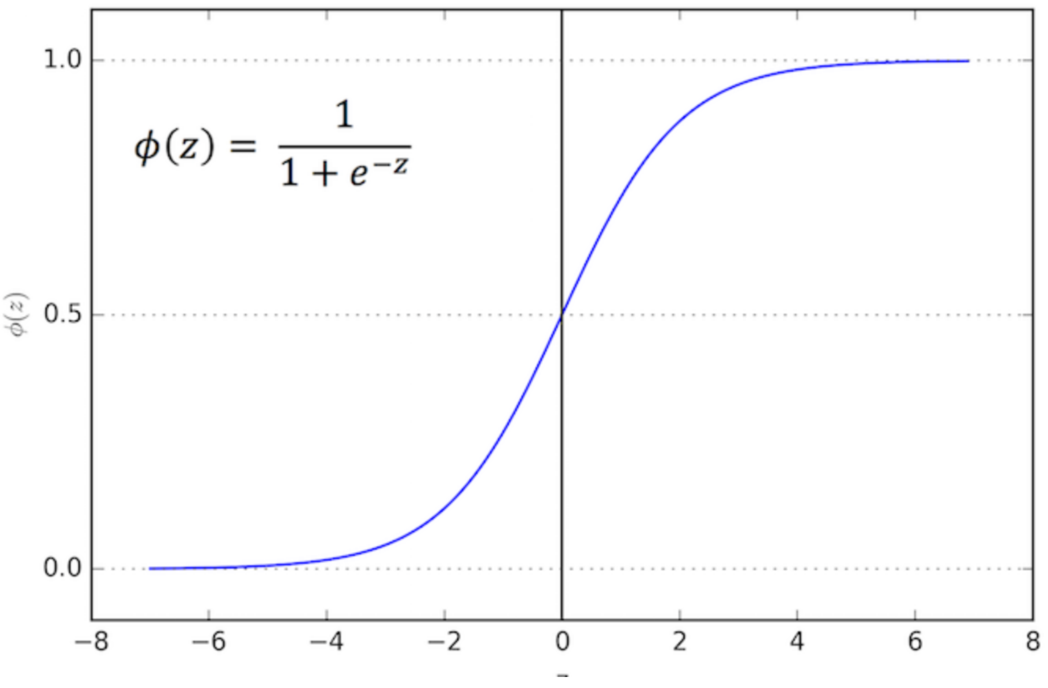

FIGURE 5.3: Logistic Regression Model

5.4.2 Multilayer Perceptron (MLP)

MLP (as depicted in figure 5.4) is a fully connected class of feed forward artificial neural network (ANN). An MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training. Its multiple layers and non-linear activation distinguish MLP from a linear perceptron. It can distinguish data that is not linearly separable. MLP is divided into an input layer, an output layer and a hidden layer. The information is collected through the input layer, and the data is input into the hidden layer for analysis and processing. This study uses a multi-layer perceptron (MLP) model with a single hidden layer, and the initial learning rate is 0.3.

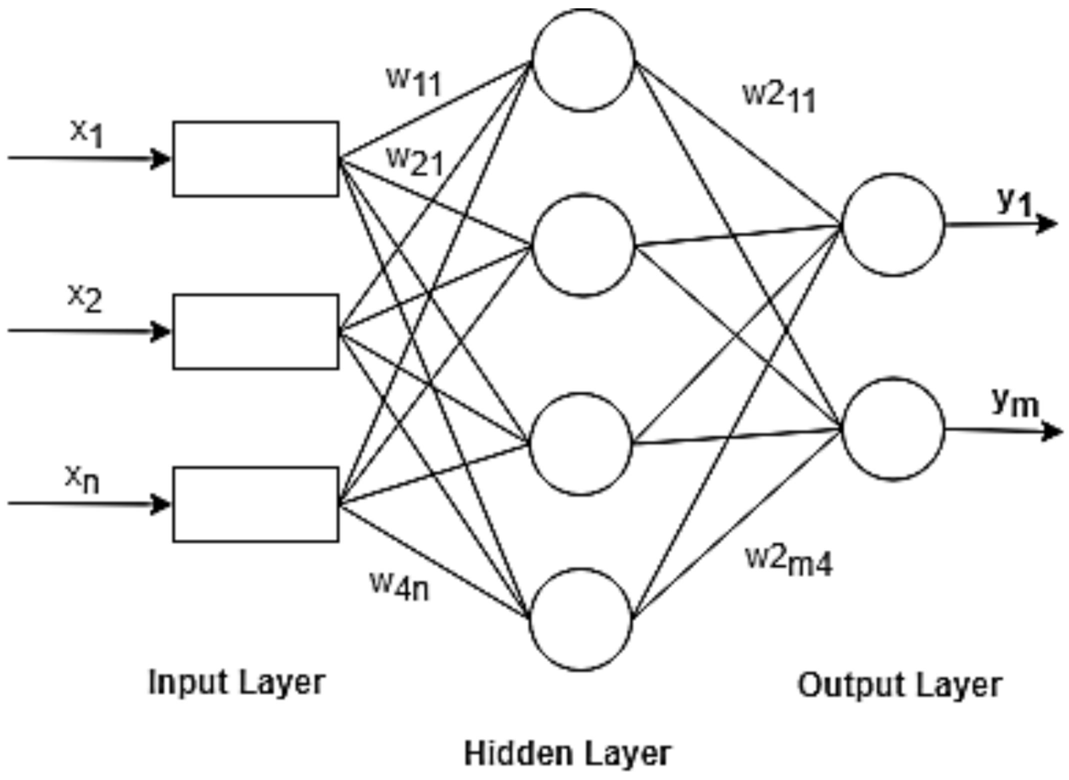

FIGURE 5.4: Multilayer Perceptron Model

5.4.3 Naive Bayes

Naive Bayes (as depicted in figure 5.5) is a simple "probabilistic classifiers" based on applying Bayes' theorem with strong independence assumptions between the features. They are among the simplest Bayesian network models, but coupled with kernel density estimation, they can achieve high accuracy levels.

Naive Bayes classifiers are highly scalable, requiring a number of parameters linear in the number of variables (features/predictors) in a learning problem.

Maximum-likelihood training can be done by evaluating a closed-form expression, which takes linear time, rather than by expensive iterative approximation as used for many other types of classifiers.

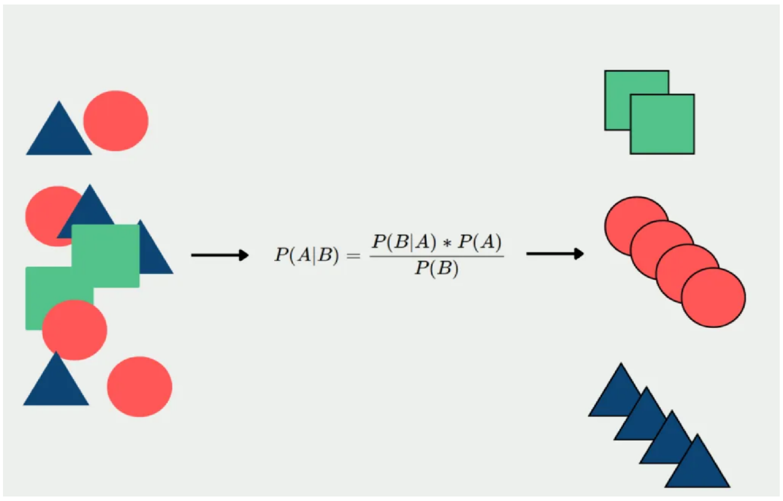

FIGURE 5.5: Naive Bayes

5.4.4 Decision Trees

Decision Trees (as depicted in figure 5.6) classifies instances by sorting them based on feature values. Each node in a decision tree represents a feature in an instance to be classified, and each branch represents a value that the node can assume. Instances are classified starting at the root node and sorted based on their feature values [98].

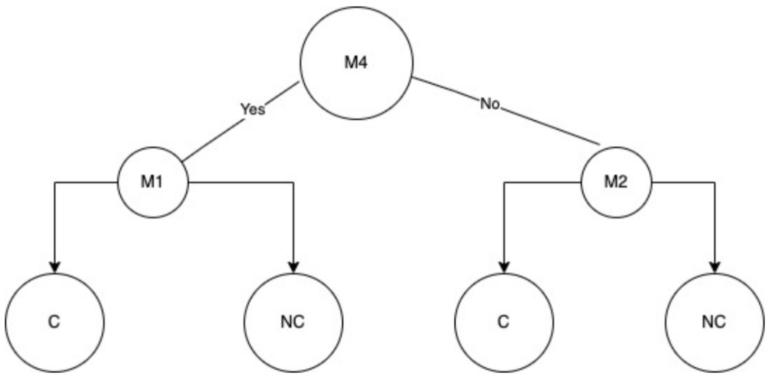

FIGURE 5.6: Decision Trees

Decision tree learning, used in data mining and machine learning, uses a decision tree as a predictive model which maps observations about an item to conclusions about the item's target value. Decision tree classifiers usually employ post-pruning techniques that evaluate the performance of decision trees, as they are pruned by using a validation set. Any node can be removed and assigned the most common class of the training instances that are sorted to it [98]. The DT-based model in this study uses the C5.0 algorithm with a minimum number of leaf nodes, which avoids the problem of too many branches in the ID3 algorithm. Also, pruning is performed during the construction of the decision tree to discretize continuous data, and the limit is set to the maximum number of leaf nodes.

5.4.5 Random Forests

Random forest (as depicted in figure 5.7) is an ensemble learning method for classification. The output of the random forest is the class selected by most trees. Random decision forests correct for decision trees' habit of overfitting to their training set. Random forests generally outperform decision trees, but their accuracy is lower than gradient boosted trees.

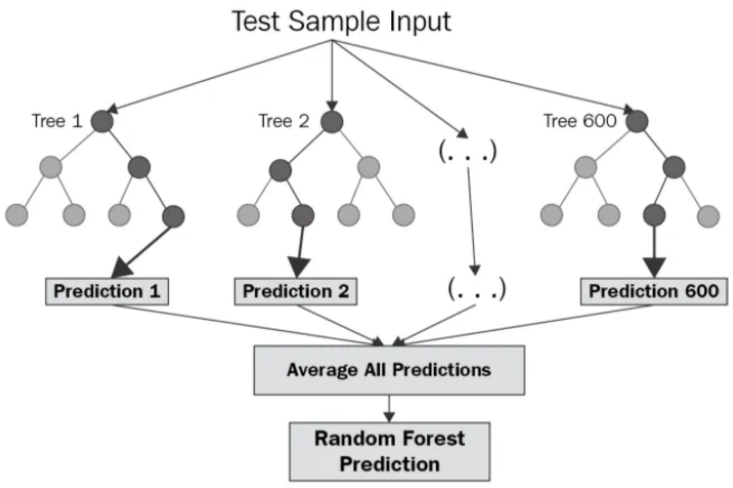

FIGURE 5.7: Random Forests

However, data characteristics can affect their performance. In a random forest (RF) model, the entire random forest is composed of 500 decision trees (ntree = 500), and each decision tree randomly selects 8 variables (mtry = 8) from the given number of variables to build a decision tree.

5.4.6 Support Vector Machines (SVM)

SVM (as depicted in figure 5.8) are closely related to classical multilayer perceptron neural networks. SVMs revolve around the notion of a margining either side of a hyperplane that separates two data classes. Maximizing the margin and thereby creating the largest possible distance between the separating hyperplane and the instances on either side of it has been proven to reduce an upper bound on the expected generalisation error [98].

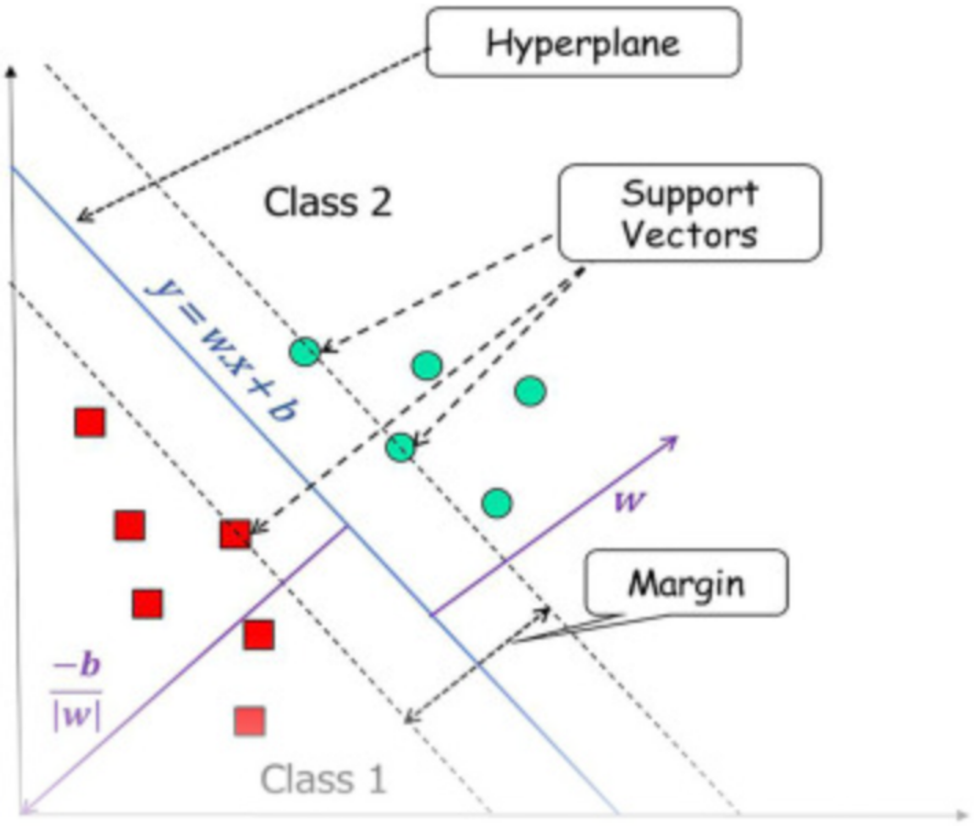

FIGURE 5.8: Support Vector Machines

SVM based model uses Gaussian inner product as the kernel function (SVM-Kernel). Through the iterative solution of sub-problems, the prediction of large-scale problems is finally completed. The gamma parameter in the model is set to 0.024.

$$y(x) = w^T\Phi(x) + c \tag{5.2}$$

where x is the input vector and y is the output vector. $\phi(x)$ is a polynomial kernel function. w and c represent the adjusted weight vector and scalar threshold values, respectively.

5.5 Cross Validation

We divided all 5 datasets into distinct training and testing subsets called folds to ensure robust training and evaluation of our machine learning models.

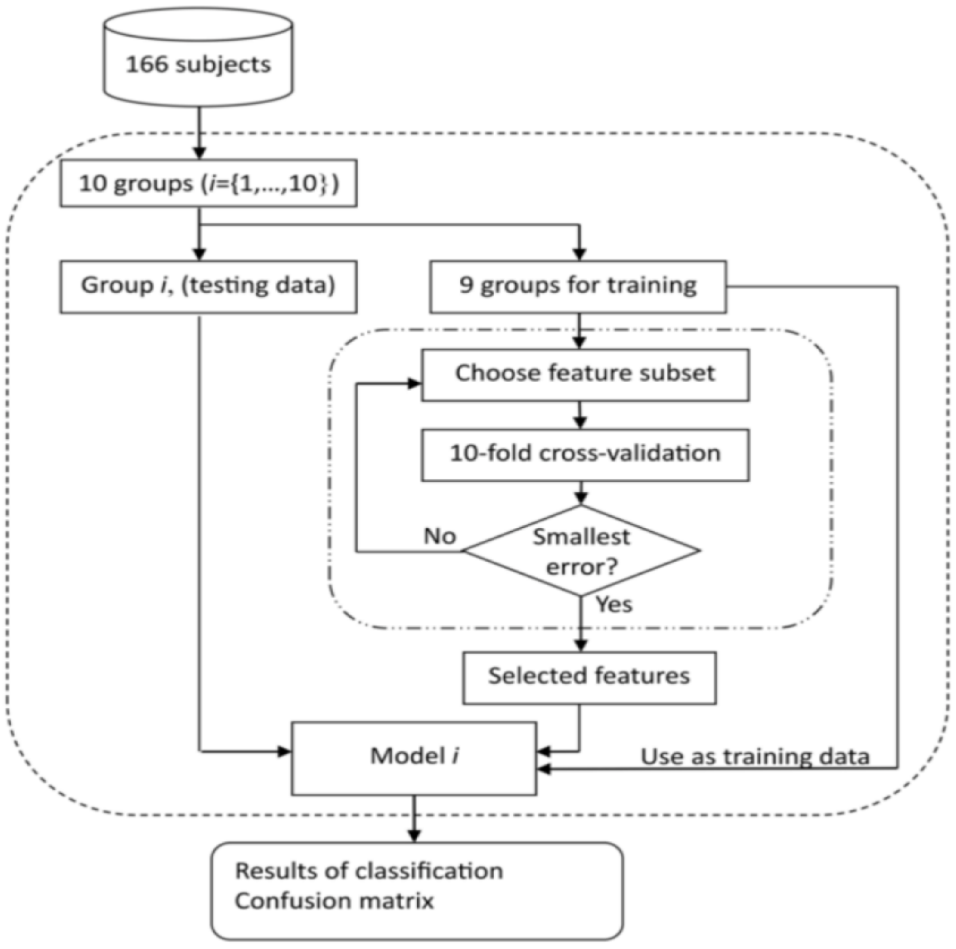

Figure 5.9: Overview of 10 fold cross validation

The datasets were partitioned using a stratified 10-fold cross-validation technique shown in figure 5.9, which ensures that each fold maintains the same class distribution as the original dataset. This approach minimizes bias and variance by cycling through different folds for training and testing, enhancing the generalization of the models. During each iteration, nine folds were used for training, while the remaining fold served as the test set. This process was repeated 20 times i.e. 20 epochs, and the results were compared by taking the confusion matrices to assess the model's performance. We choose folds each time for each epoch randomly. Python using Scikit-learn is used for the implementation to facilitate the seamless implementation of this partitioning and ensure consistency across all machine-learning algorithms applied to the datasets.

5.6 Evaluation Measure

For evaluation purposes, we adopt three commonly used performance measures identified in our literature review: Accuracy, F-measure, and AUC.

Accuracy shows the correct predictions. It is a good measure when the classes in the test dataset are nearly balanced. It measures the ability of a classifier in correctly identifying all samples, no matter it is positive or negative.

$$Accuracy = \frac{TP + TN}{P + N} \tag{5.3}$$

The F-measure is a measure of the model's accuracy, which considers both precision and recall. It is known as the harmonic mean of precision and recall. F-measure is obtained using Equation 5.4. The values of the F-measure range from 0 to 1; the value of 1 indicates perfect precision and recall, and a value of 0 indicates that either precision or recall is 0.

$$Fmeasure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{5.4}$$

AUC (Area under the Receiver Operating Characteristics Curve) is a probability curve and represents the degree or measure of separability. It tells how much the model is capable of distinguishing between the classes. Values of AUC lie in the range of 0 to 1. An AUC of 0.0 represents 100% wrong prediction, and an AUC of 1.0 represents 100% correct prediction.

To ensure thorough model evaluation, the chosen performance metrics F1-score, Accuracy, and AUC were properly justified. The F1-score was selected because it offers a balanced evaluation of false positives and false negatives and is robust when dealing with imbalanced datasets, where a high accuracy may be deceptive due to an excessive number of non faulty modules. In order to provide a baseline understanding of performance across all forecasts, accuracy was added as a general metric of overall model correctness. The model's discriminative power across different classification thresholds was assessed using AUC, which makes it appropriate for model comparisons without regard to particular decision boundaries. When taken as a whole, these metrics guarantee a comprehensive evaluation of the models, encompassing not only their overall performance but also their capacity to appropriately balance errors and differentiate across classes. Finally, 10-fold cross-validation is used to validate the performance of the prediction models. In cross-validation, the input data set is randomly partitioned into 10 folds of equal size. 9 folds are used to train the model and the remaining 1 fold is used to test the model. This procedure is repeated 10 times, each time leaving out a different fold for testing.

5.7 Results and Discussion

In this section, we present the results obtained from four distinct experiments, namely Experiment-1, Experiment-2, Experiment-3, and Experiment-4. These experiments aimed to explore the impact of decomposing Halstead metrics on the performance of fault prediction models. Specifically, we delved into the decomposition of Halstead metrics for both operators and operands, spanning various levels of granularity. Our comprehensive analysis involved meticulously examining the experimental results derived from these distinct settings. Through rigorous experimentation and meticulous analysis, we sought to ascertain the optimal levels of decomposition that yield the most favorable outcomes for fault prediction tasks. Our findings indicate that, in the case of operator decomposition, the most favorable results were achieved at level 1. Conversely, for operand decomposition, optimal outcomes were observed at level 1. These conclusions are drawn based on a thorough evaluation of prediction performance of decomposed Halstead base metrics across different levels of decomposition. Moreover, to supplement our findings, we conducted a comparative analysis with previous research outcomes, providing further insights into the effectiveness and significance of our experimental approach. Furthermore, we have meticulously presented our experimental findings using tabular and graphical representations in the subsequent sections, facilitating a comprehensive understanding and interpretation of the results obtained.

5.7.1 Experimental Results with Decomposed Halstead Operators at Level 1

The table 5.5, 5.6, and 5.7 presents the results of two experiments, Exp1 and Exp2, show how well machine learning models perform when employing decomposed Halstead operators at Level 1 in Experiment 2 and conventional Halstead metrics in Experiment 1.

Table 5.5: Accuracy in Experiment 1 and 2

Dataset	LR		NB		DT		MLP		RF		SVM	
	Exp1	Exp2	Exp1	Exp2	Exp1	Exp2	Exp1	Exp2	Exp1	Exp2	Exp1	Exp2
Apache Lucene 2.4	0.75	0.79	0.66	0.77	0.80	0.88	0.70	0.80	0.76	0.85	0.82	0.89
Eclipse equinox framework 3.4	0.77	0.83	0.72	0.77	0.69	0.80	0.65	0.78	0.76	0.79	0.79	0.84
Eclipse JDT Core 3.4	0.73	0.83	0.68	0.78	0.73	0.82	0.79	0.80	0.67	0.80	0.81	0.89
Eclipse PDE UI 3.4.1	0.72	0.80	0.79	0.88	0.65	0.80	0.69	0.80	0.65	0.77	0.76	0.82
Mylyn 3.1	0.66	0.75	0.8	0.87	0.72	0.79	0.66	0.74	0.73	0.78	0.76	0.82
Average	0.71	0.79	0.73	0.82	0.72	0.82	0.70	0.79	0.71	0.80	0.79	0.85

TABLE 5.6: F-measure in Experiment 1 and 2

Dataset	LR		NB		DT		MLP		RF		SVM	
	Exp1	Exp2	Exp1	Exp2	Exp1	Exp2	Exp1	Exp2	Exp1	Exp2	Exp1	Exp2
Apache Lucene 2.4	0.78	0.82	0.76	0.82	0.77	0.82	0.74	0.83	0.69	0.78	0.80	0.88
Eclipse equinox framework 3.4	0.70	0.81	0.72	0.84	0.75	0.84	0.75	0.84	0.75	0.86	0.79	0.89
Eclipse JDT Core 3.4	0.72	0.84	0.79	0.87	0.68	0.78	0.70	0.79	0.69	0.80	0.82	0.91
Eclipse PDE UI 3.4.1	0.74	0.83	0.82	0.90	0.69	0.82	0.76	0.82	0.67	0.81	0.78	0.86
Mylyn 3.1	0.74	0.81	0.81	0.90	0.70	0.82	0.73	0.80	0.71	0.82	0.79	0.84
Average	0.74	0.82	0.78	0.86	0.72	0.82	0.74	0.82	0.70	0.81	0.80	0.87

Experiment 2 consistently performs better than Experiment 1 across the three main assessment metrics of Accuracy, F-measure, and AUC, according to the experimental results shown in Tables 5.5, 5.6, and 5.7 as well as Figures 5.10, 5.11, and 5.12.

TABLE 5.7: AUC in Experiment 1 and 2

Dataset	LR		NB		DT		MLP		RF		SVM	
	Exp1	Exp2	Exp1	Exp2	Exp1	Exp2	Exp1	Exp2	Exp1	Exp2	Exp1	Exp2
Apache Lucene 2.4	0.79	0.87	0.71	0.78	0.66	0.75	0.77	0.87	0.74	0.78	0.82	0.91
Eclipse equinox framework 3.4	0.74	0.84	0.67	0.75	0.68	0.81	0.69	0.77	0.71	0.83	0.79	0.88
Eclipse JDT Core 3.4	0.74	0.81	0.70	0.79	0.73	0.83	0.73	0.80	0.76	0.84	0.81	0.88
Eclipse PDE UI 3.4.1	0.77	0.81	0.81	0.89	0.73	0.80	0.77	0.82	0.66	0.76	0.73	0.81
Mylyn 3.1	0.68	0.77	0.79	0.89	0.73	0.83	0.80	0.88	0.67	0.76	0.68	0.78
Average	0.74	0.82	0.74	0.82	0.71	0.81	0.75	0.83	0.71	0.79	0.77	0.85

With regard to accuracy, the SVM classifier provided the highest average result in Experiment 2, with 0.85, as opposed to 0.79 in Experiment 1. This enhancement shows how decomposition of operator at level 1 features can help the classifier identify intricate patterns in the data.

Similarly, the F-measure results in Table 5.6 show a significant increase, with the highest average F-measure in Experiment 2 also achieved by the SVM classifier, reaching 0.87, compared to 0.80 in Experiment 1. This indicates that the models in Experiment 2 not only make more accurate predictions but also reduce false positives and false negatives more effectively. The Decision Tree (DT) and Random

Forest (RF) classifiers also show notable improvements in their F-measure values, reinforcing the conclusion that decomposed operators at level 1 enhance the balance between precision and recall.

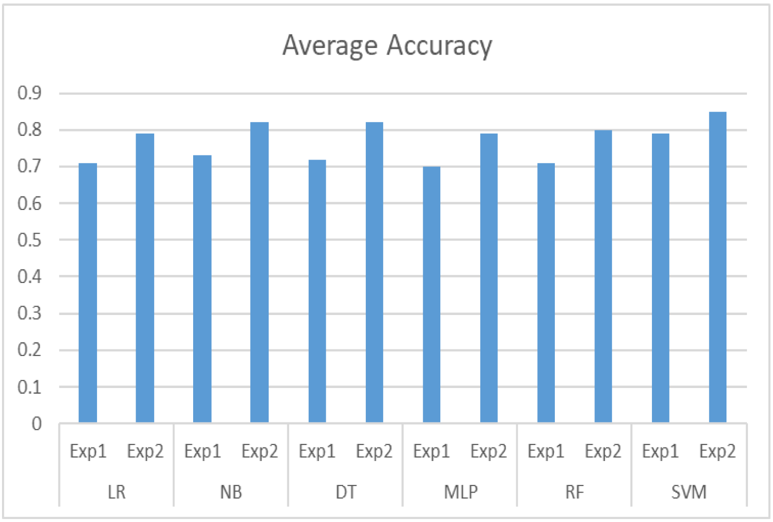

FIGURE 5.10: Difference in average accuracy regarding selected datasets against ML algorithms in Exp1 and Exp2.

The AUC results in Table 5.7 further confirm the superior classification capability of Experiment 2. The highest average AUC score, achieved by the SVM model, is 0.85 in Experiment 2, compared to 0.77 in Experiment 1.

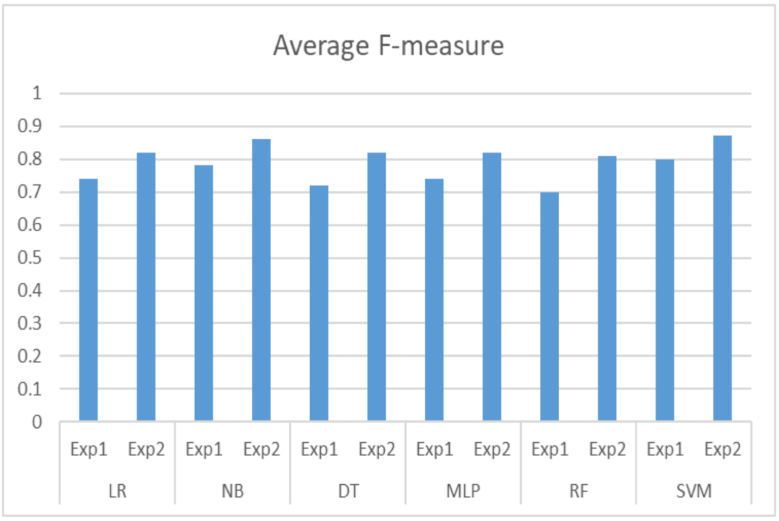

FIGURE 5.11: Difference in average F-measure regarding selected datasets against ML algorithms in Exp1 and Exp2.

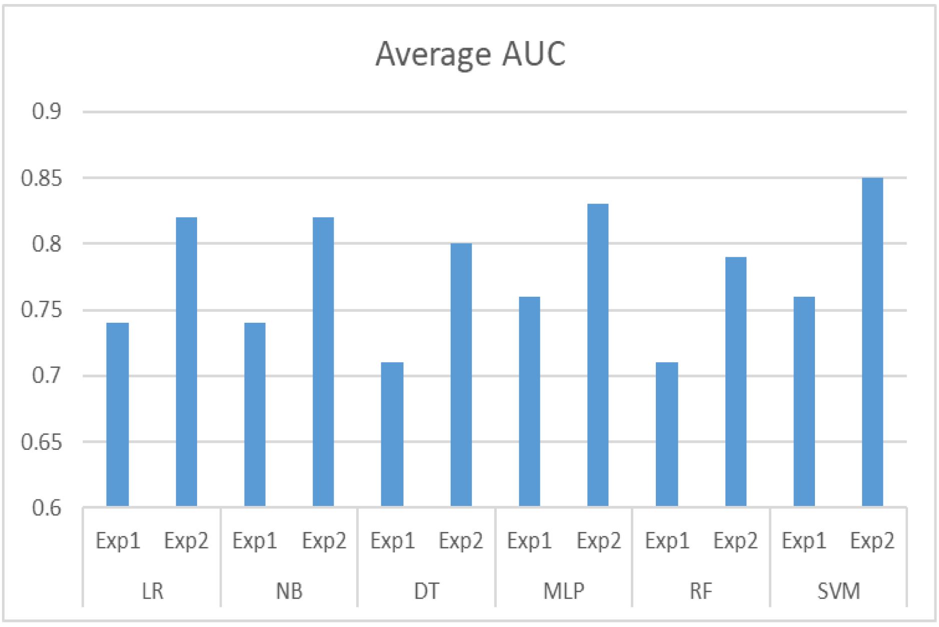

FIGURE 5.12: Difference in average AUC regarding selected datasets against ML algorithms in Exp1 and Exp2.

This increase suggests that the decomposed metrics provide a richer and more informative feature set, enabling the classifiers to better distinguish between fault-prone and non-fault-prone modules. Logistic Regression also benefits from this added granularity, achieving an AUC score of 0.82 in Experiment 2 compared to 0.74 in Experiment 1. Overall, the findings highlight that more reliable and accurate software fault predictions are produced by decomposed Halstead operators at Level 1. The idea that more feature granularity improves machine learning models' predictive ability is supported by the better performance on all metrics, especially SVM. In order to increase software dependability and testing efficiency, these results lend credence to the use of deconstructed metrics in fault prediction systems.

5.7.2 Experimental Results with Decomposed Halstead Operators and Operands at Level 1

Machine learning models utilizing decomposed Halstead operators at Level 1 in Experiment 2 and both decomposed Halstead operators and operands at Level 1 in Experiment 3 are compared in the results in this section. Accuracy, F-measure, and AUC are the three main metrics that are compared in Tables 5.8, 5.9, and 5.10 as well as Figures 5.13, 5.14, and 5.15. In Experiment 3 addition of decomposed operands improves feature granularity even further, leading to better performance across the majority of classifiers as compare to Experiment 2 having decomposed operators at level 1.

Table 5.8: Accuracy in Experiment 2 and 3

Dataset	LR		NB		DT		MLP		RF		SVM	
	Exp2	Exp3	Exp2	Exp3	Exp2	Exp3	Exp2	Exp3	Exp2	Exp3	Exp2	Exp3
Apache Lucene 2.4	0.87	0.94	0.78	0.85	0.75	0.84	0.87	0.96	0.78	0.81	0.91	0.99
Eclipse equinox framework 3.4	0.77	0.83	0.77	0.82	0.80	0.91	0.78	0.91	0.79	0.83	0.84	0.89
Eclipse JDT Core 3.4	0.83	0.93	0.78	0.89	0.82	0.91	0.80	0.82	0.80	0.94	0.89	0.97
Eclipse PDE UI 3.4.1	0.80	0.88	0.88	0.98	0.80	0.96	0.80	0.92	0.77	0.90	0.82	0.87
Mylyn 3.1	0.75	0.83	0.87	0.95	0.79	0.87	0.74	0.82	0.78	0.84	0.82	0.87
Average	0.79	0.86	0.82	0.90	0.82	0.91	0.79	0.88	0.80	0.89	0.85	0.91

Table 5.9: F-measure in Experiment 2 and 3

Dataset	LR		NB		DT		MLP		RF		SVM	
	Exp2	Exp3	Exp2	Exp3	Exp2	Exp3	Exp2	Exp3	Exp2	Exp3	Exp2	Exp3
Apache Lucene 2.4	0.82	0.86	0.82	0.87	0.82	0.87	0.83	0.91	0.78	0.86	0.88	0.96
Eclipse equinox framework 3.4	0.81	0.91	0.84	0.95	0.84	0.92	0.83	0.91	0.86	0.95	0.89	0.98
Eclipse JDT Core 3.4	0.84	0.96	0.87	0.95	0.78	0.88	0.79	0.89	0.80	0.91	0.91	0.99
Eclipse PDE UI 3.4.1	0.83	0.91	0.90	0.98	0.82	0.95	0.82	0.88	0.81	0.95	0.86	0.93
Mylyn 3.1	0.81	0.87	0.90	0.99	0.82	0.93	0.80	0.87	0.82	0.92	0.84	0.88
Average	0.82	0.91	0.86	0.95	0.82	0.91	0.82	0.89	0.81	0.92	0.87	0.95

Table 5.10: AUC in Experiment 2 and 3

Dataset	LR		NB		DT		MLP		RF		SVM	
	Exp2	Exp3	Exp2	Exp3	Exp2	Exp3	Exp2	Exp3	Exp2	Exp3	Exp2	Exp3
Apache Lucene 2.4	0.87	0.94	0.78	0.85	0.75	0.84	0.87	0.96	0.78	0.81	0.91	0.99
Eclipse equinox framework 3.4	0.84	0.94	0.75	0.83	0.81	0.93	0.77	0.84	0.83	0.94	0.88	0.97
Eclipse JDT Core 3.4	0.81	0.88	0.79	0.88	0.83	0.93	0.80	0.86	0.84	0.91	0.88	0.95
Eclipse PDE UI 3.4.1	0.81	0.84	0.89	0.96	0.80	0.86	0.82	0.87	0.76	0.86	0.81	0.88
Mylyn 3.1	0.77	0.86	0.89	0.99	0.83	0.92	0.88	0.96	0.76	0.85	0.78	0.87
Average	0.82	0.89	0.82	0.90	0.80	0.90	0.83	0.90	0.79	0.87	0.85	0.93

The accuracy findings are displayed in Table 5.8, which demonstrates that, for all classifiers, Experiment 3 obtains a better average accuracy than Experiment 2.

With an average accuracy of 0.91 in Experiment 3 against 0.85 in Experiment 2, the SVM classifier once again performs best.

The advantage of including both decomposed operators and operands is highlighted by this improvement, which enables the model to capture more intricate structural elements of the code. With an accuracy of 0.89 in Experiment 3 as opposed to 0.80 in Experiment 2, RF likewise shows improvement.

The benefit of Experiment 3 is further supported by the F-measure values in Table 5.9. While Experiment 2's average F-measure was 0.87, the SVM classifier achieves the greatest average F-measure of 0.95.

Notable gains are also demonstrated by other classifiers, like Decision Trees and Multilayer Perceptron (MLP), demonstrating the harmony between recall and precision attained when employing both decomposed operators and operands.

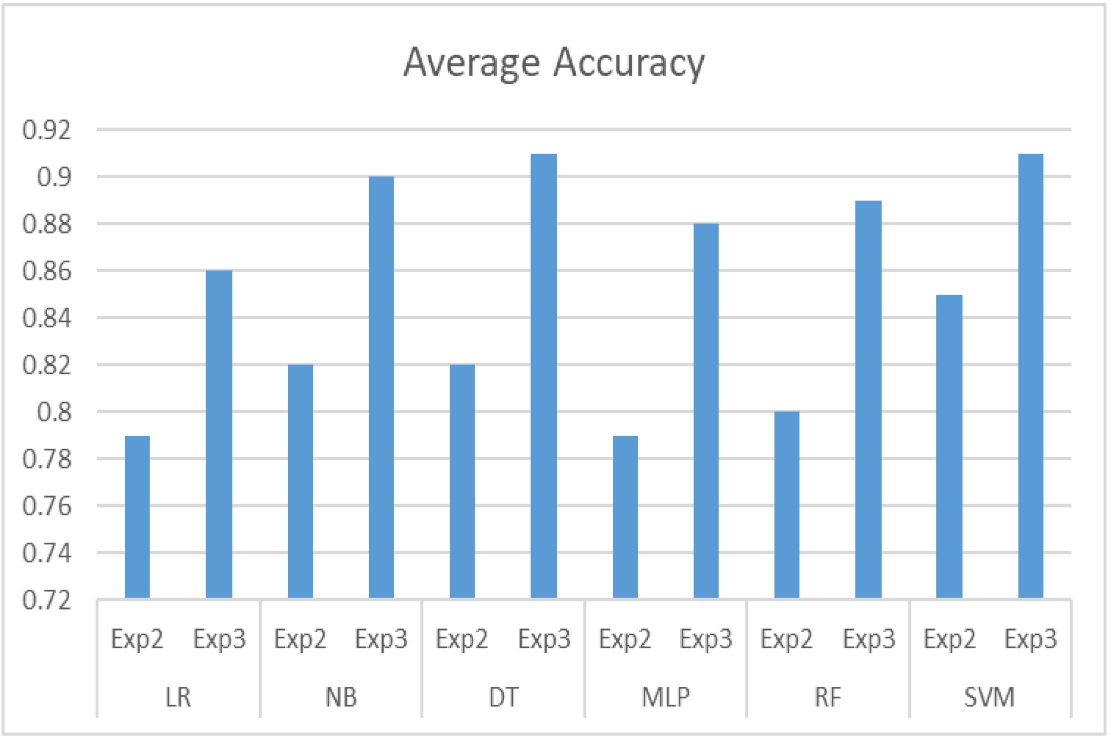

FIGURE 5.13: Difference in average accuracy regarding selected datasets against ML algorithms in Exp2 and Exp3.

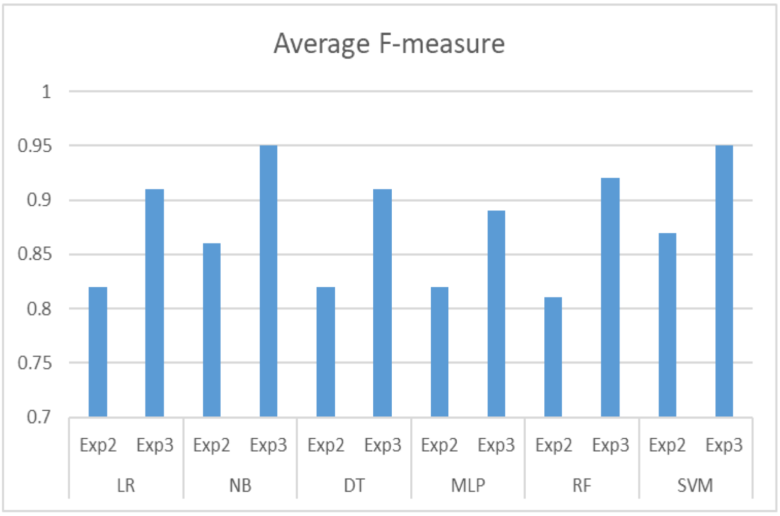

FIGURE 5.14: Difference in average F-measure regarding selected datasets against ML algorithms in Exp2 and Exp3.

This suggests a decrease in the frequencies of false positives and false negatives, increasing the accuracy of the forecasts.

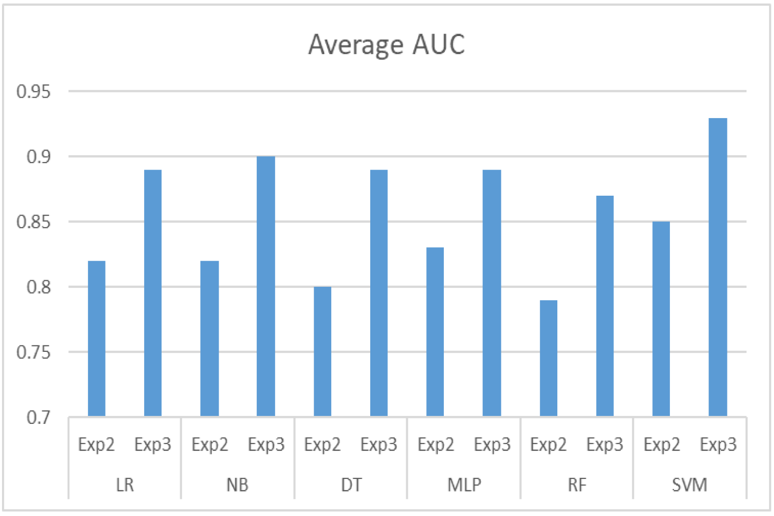

FIGURE 5.15: Difference in average AUC regarding selected datasets against ML algorithms in Exp2 and Exp3.

The AUC scores in Experiment 3 across several classifiers demonstrate notable increases, as seen in Table 5.10. With an AUC of 0.93 as opposed to 0.85 in Experiment 2, the SVM model shows a greater capacity to differentiate between modules that are prone to faulty and those that are not.

The RF classifier also benefits, improving its AUC from 0.79 in Experiment 2 to 0.87 in Experiment 3. Figures 5.13, 5.14, and 5.15 illustrate these trends visually, showing a consistent increase in average performance across accuracy, F-measure, and AUC for all classifiers when operands are decomposed alongside operators.

Consistent improvements in Experiment 3 across accuracy, F-measure, and AUC validate the hypothesis that increasing feature granularity improves prediction performance.

The results show that decomposing both operators and operands at Level 1 gives machine learning models a richer and more informative feature set, which in turn improves fault prediction capabilities.

Combining operator and operand decomposition effectively achieves a more comprehensive representation of code complexity and structure, which in turn improves the robustness of software fault prediction frameworks.

5.7.3 Experimental Results with Decomposed Halstead Operators at Level 2

The results in this section compare the performance of machine learning models using decomposed Halstead operators and operands at Level 1 in Experiment 3 and decomposed Halstead operators at Level 2 in Experiment 4. Tables 5.11, 5.12, and 5.13, along with Figures 5.16, 5.17, and 5.18, demonstrate the performance trends across key performance measure accuracy, F-measure, and AUC.

TABLE 5.11: Accuracy in Experiment 3 and 4

Dataset	LR		NB		DT		MLP		RF		SVM	
	Exp3	Exp4	Exp3	Exp4	Exp3	Exp4	Exp3	Exp4	Exp3	Exp4	Exp3	Exp4
Apache Lucene 2.4	0.83	0.70	0.88	0.70	0.91	0.79	0.91	0.71	0.94	0.72	0.97	0.79
Eclipse equinox framework 3.4	0.83	0.68	0.82	0.69	0.91	0.70	0.91	0.60	0.83	0.71	0.89	0.69
Eclipse JDT Core 3.4	0.93	0.75	0.89	0.60	0.91	0.75	0.82	0.70	0.94	0.69	0.97	0.78
Eclipse PDE UI 3.4.1	0.88	0.77	0.98	0.68	0.96	0.71	0.92	0.68	0.90	0.71	0.87	0.77
Mylyn 3.1	0.83	0.70	0.95	0.84	0.87	0.69	0.82	0.60	0.84	0.70	0.87	0.71
Average	0.86	0.72	0.90	0.70	0.91	0.73	0.88	0.66	0.89	0.71	0.91	0.75

TABLE 5.12: F-measure in Experiment 3 and 4

Dataset	LR		NB		DT		MLP		RF		SVM	
	Exp3	Exp4	Exp3	Exp4	Exp3	Exp4	Exp3	Exp4	Exp3	Exp4	Exp3	Exp4
Apache Lucene 2.4	0.86	0.75	0.87	0.71	0.87	0.75	0.91	0.73	0.86	0.71	0.96	0.76
Eclipse equinox framework 3.4	0.91	0.69	0.95	0.68	0.92	0.72	0.92	0.69	0.96	0.73	0.98	0.75
Eclipse JDT Core 3.4	0.96	0.68	0.95	0.72	0.88	0.65	0.89	0.66	0.91	0.65	0.99	0.75
Eclipse PDE UI 3.4.1	0.91	0.71	0.98	0.78	0.95	0.66	0.88	0.71	0.95	0.65	0.93	0.73
Mylyn 3.1	0.87	0.69	0.99	0.77	0.93	0.69	0.87	0.69	0.92	0.69	0.88	0.71
Average	0.91	0.70	0.95	0.73	0.91	0.69	0.89	0.70	0.92	0.69	0.95	0.74

The results indicate a noticeable decline in performance when moving from Level 1 to Level 2 decomposition for operators, emphasizing that the optimal granularity for software fault prediction is achieved at Level 1 decomposition of both operators and operands.

In Table 5.11, the accuracy scores show that Experiment 4 generally performs worse compared to Experiment 3 across most classifiers.

The SVM classifier, which previously achieved the highest average accuracy of 0.91 in Experiment 3, drops to 0.75 in Experiment 4.

Similarly, the accuracy of Logistic Regression decreases from 0.86 to 0.72. This decline indicates that further decomposition at Level 2 introduces noise rather than improving the predictive power, as the increased number of features may lead to over fitting or redundancy in the models.

TABLE 5.13: AUC in Experiment 3 and 4

Dataset	LR		NB		DT		MLP		RF		SVM	
	Exp3	Exp4	Exp3	Exp4	Exp3	Exp4	Exp3	Exp4	Exp3	Exp4	Exp3	Exp4
Apache Lucene 2.4	0.94	0.78	0.85	0.67	0.84	0.62	0.96	0.68	0.81	0.69	0.99	0.81
Eclipse equinox framework 3.4	0.94	0.71	0.83	0.69	0.93	0.59	0.84	0.67	0.94	0.68	0.97	0.69
Eclipse JDT Core 3.4	0.88	0.69	0.88	0.62	0.93	0.69	0.86	0.74	0.91	0.75	0.95	0.79
Eclipse PDE UI 3.4.1	0.84	0.73	0.96	0.78	0.86	0.69	0.87	0.72	0.86	0.65	0.88	0.69
Mylyn 3.1	0.86	0.71	0.99	0.68	0.92	0.62	0.96	0.79	0.85	0.63	0.67	0.65
Average	0.89	0.72	0.90	0.67	0.90	0.64	0.90	0.72	0.87	0.68	0.93	0.73

Table 5.12 presents the F-measure results, which follow a similar trend. The SVM classifier, which had an F-measure of 0.95 in Experiment 3, drops to 0.74 in Experiment 4. Other classifiers, such as Decision Trees and MLP, also exhibit noticeable decreases in their F-measure values. The decline in F-measure indicates that the models are struggling to maintain a balance between precision and recall when operators are further decomposed at Level 2, leading to an increase in both false positives and false negatives. In Table 5.13, the AUC values also decrease across all classifiers in Experiment 4. The Random Forest model's AUC drops from 0.87 in Experiment 3 to 0.68 in Experiment 4, and the SVM classifier's AUC decreases from 0.93 to 0.73. These decreases highlight that Level 2 decomposition does not provide meaningful additional information to distinguish between fault-prone and non-fault-prone modules. Figures 5.16, 5.17, and 5.18 further illustrate this trend, with the average performance curves for accuracy, F-measure, and AUC showing a clear decline when Level 2 decomposition is applied.

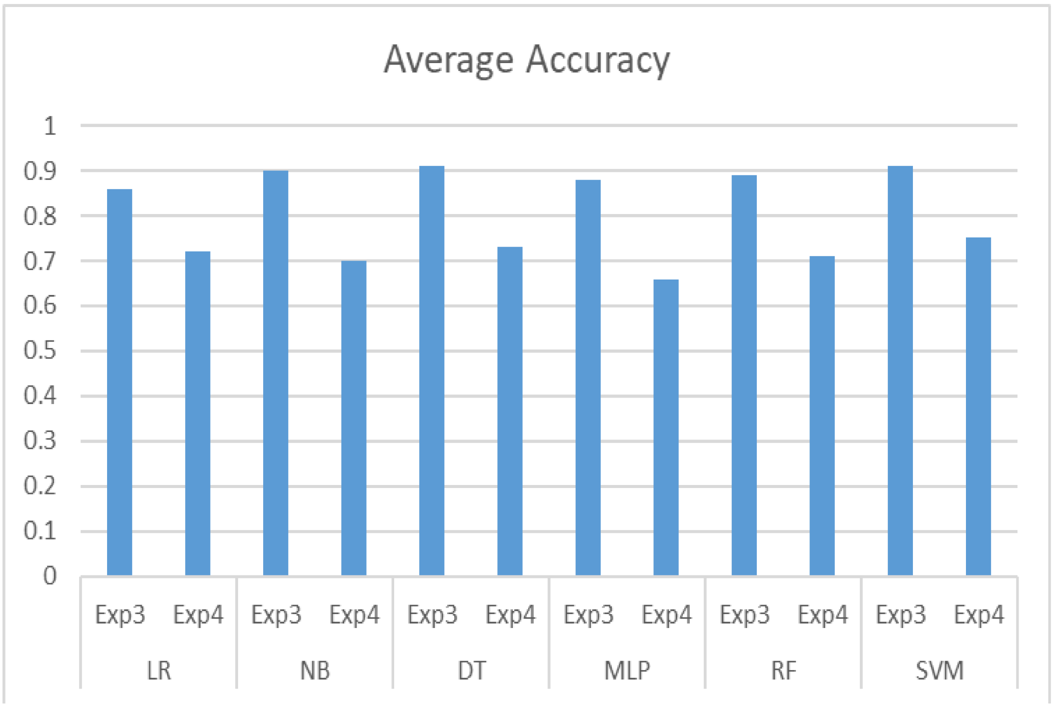

FIGURE 5.16: Difference in average accuracy regarding selected datasets against ML algorithms in Exp3 and Exp4.

Overall, the findings highlight that the optimal granularity for fault prediction is provided by decomposition of Halstead operators and operands at Level 1, which captures pertinent feature interactions without adding undue complexity.

The diminishing rewards of deeper decomposition levels are highlighted by the performance drop seen in Experiment 4. These results confirm that Level 1 decomposition is the best method for software fault prediction tasks because it achieves the ideal balance between feature richness and model interpretability.

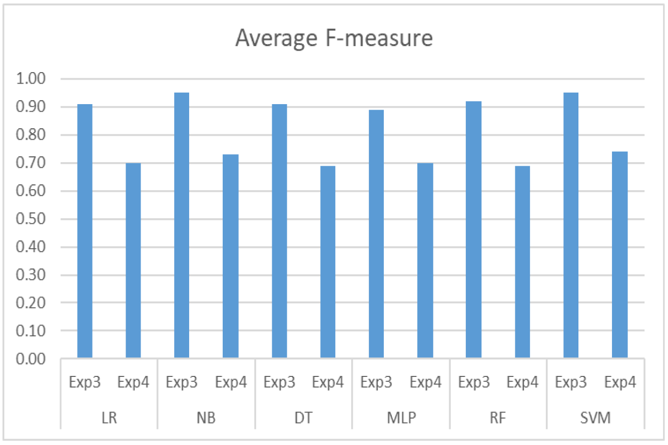

FIGURE 5.17: Difference in average F-measure regarding selected datasets against ML algorithms in Exp3 and Exp4.

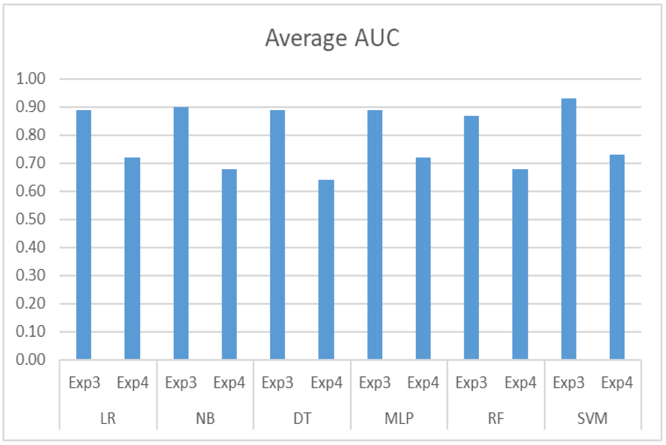

FIGURE 5.18: Difference in average AUC regarding selected datasets against ML algorithms in Exp3 and Exp4.

5.7.4 Combine Experimental Results

The combined experimental results from Experiments 1, 2, 3, and 4 are summarized in this section, which also shows the relative performance of decomposed operators at Level 1, decomposed operators and operands at Level 1, and traditional Halstead metrics.

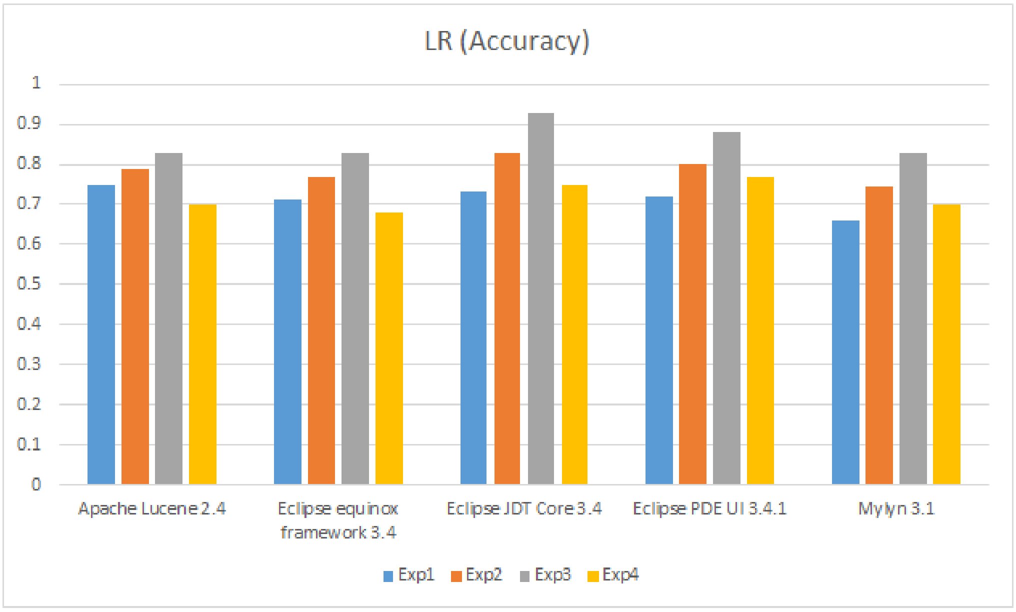

FIGURE 5.19: Difference in Accuracy with LR by introducing decomposed Halstead.

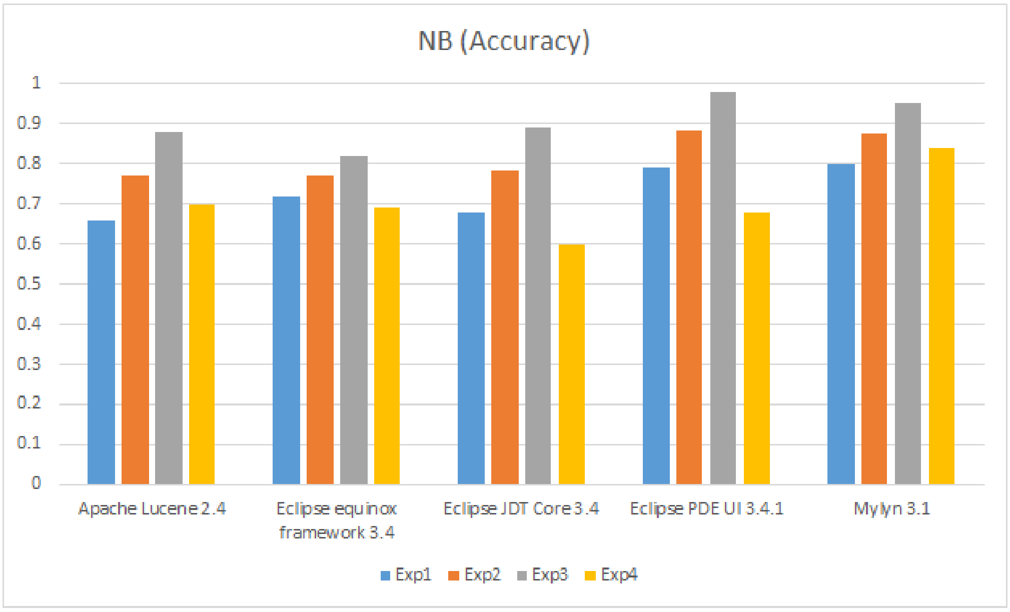

FIGURE 5.20: Difference in Accuracy with NB by introducing decomposed Halstead.

A thorough visual depiction of the effects of these various settings on the accuracy, F-measure, and AUC metrics of machine learning models can be found in the graphs from Figures 5.19 to 5.36.

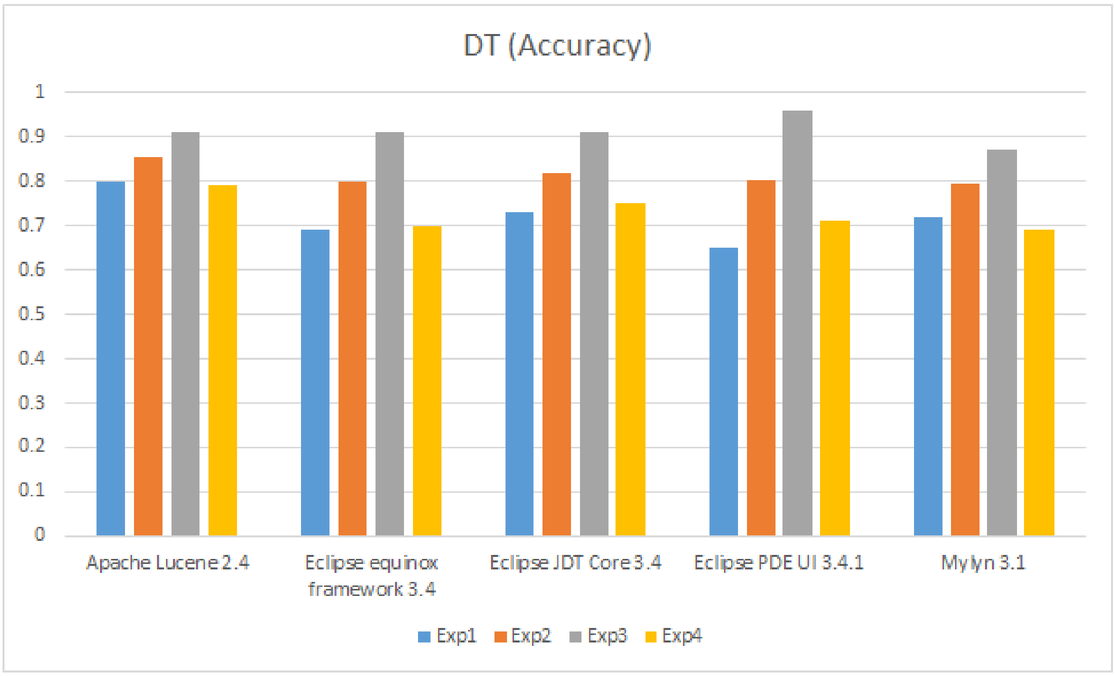

FIGURE 5.21: Difference in Accuracy with DT by introducing decomposed Halstead.

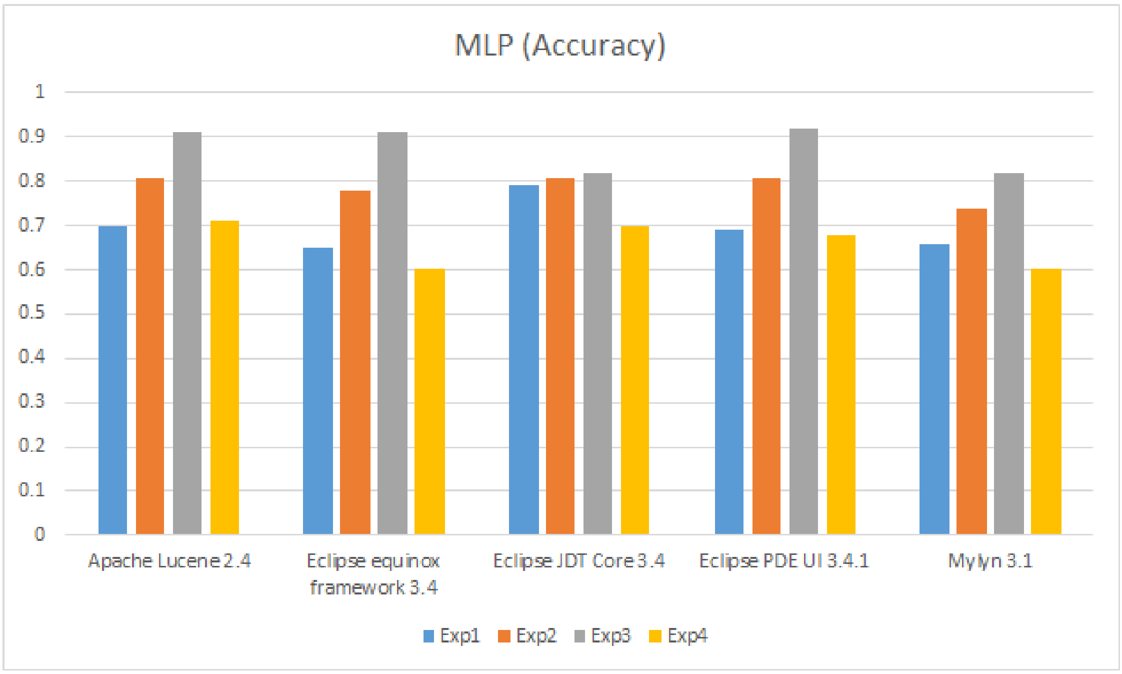

FIGURE 5.22: Difference in Accuracy with MLP by introducing decomposed Halstead.

The general pattern demonstrates that the decomposition of both operators and operands at Level 1 in Experiment 3 performs the best predictive performance, whereas further decomposition at Level 2 results in worse outcomes.

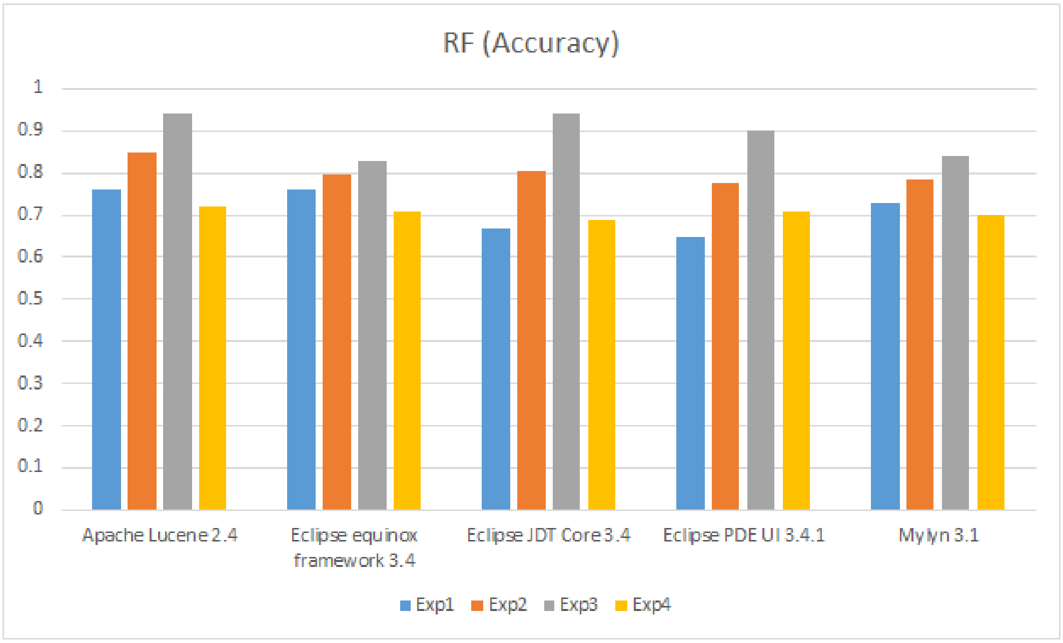

FIGURE 5.23: Difference in Accuracy with RF by introducing decomposed Halstead.

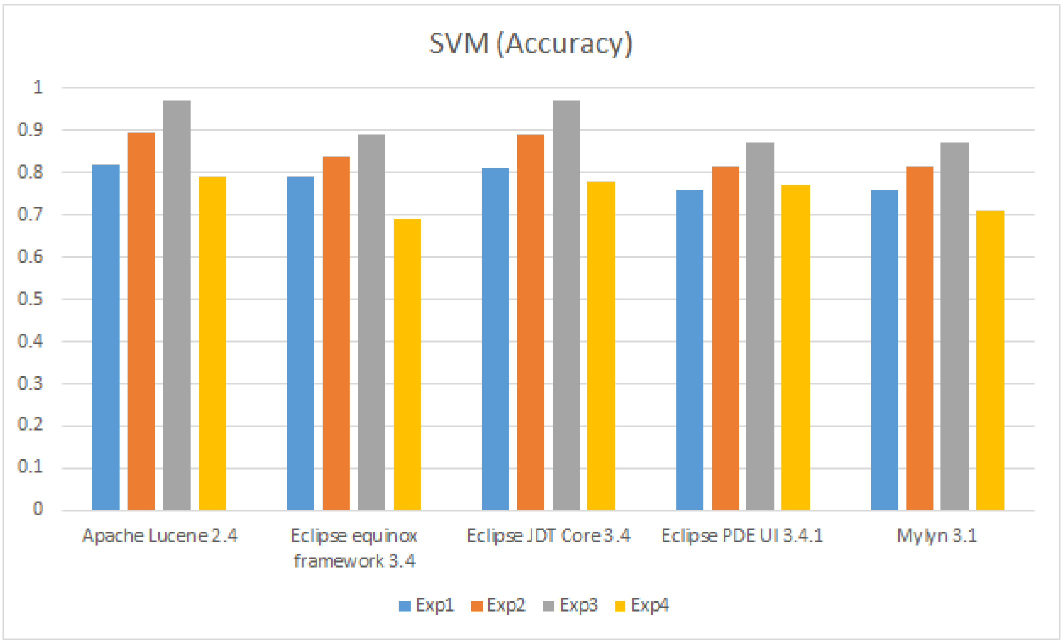

FIGURE 5.24: Difference in Accuracy with SVM by introducing decomposed Halstead.

Figures 5.19 to 5.24 accuracy trends demonstrate that while decomposition of operators alone in Experiment 2 outperforms as compere to traditional Halstead metrics in Experiment 1, the addition of decomposed operands in Experiment 3 greatly increases the predictive power of the models.

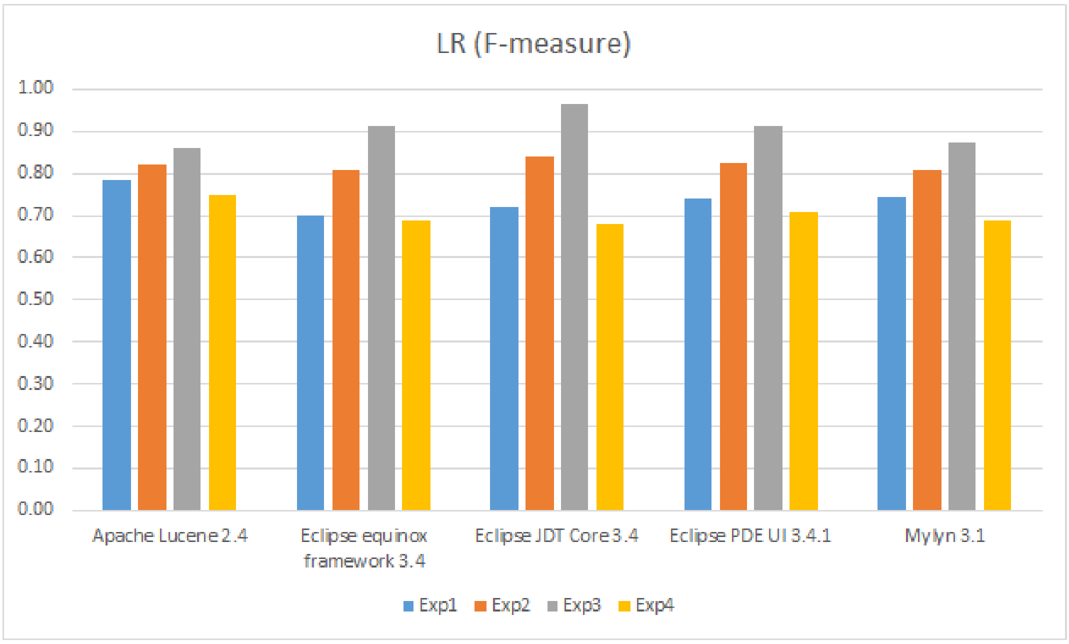

FIGURE 5.25: Difference in F-measure with LR by introducing decomposed Halstead.

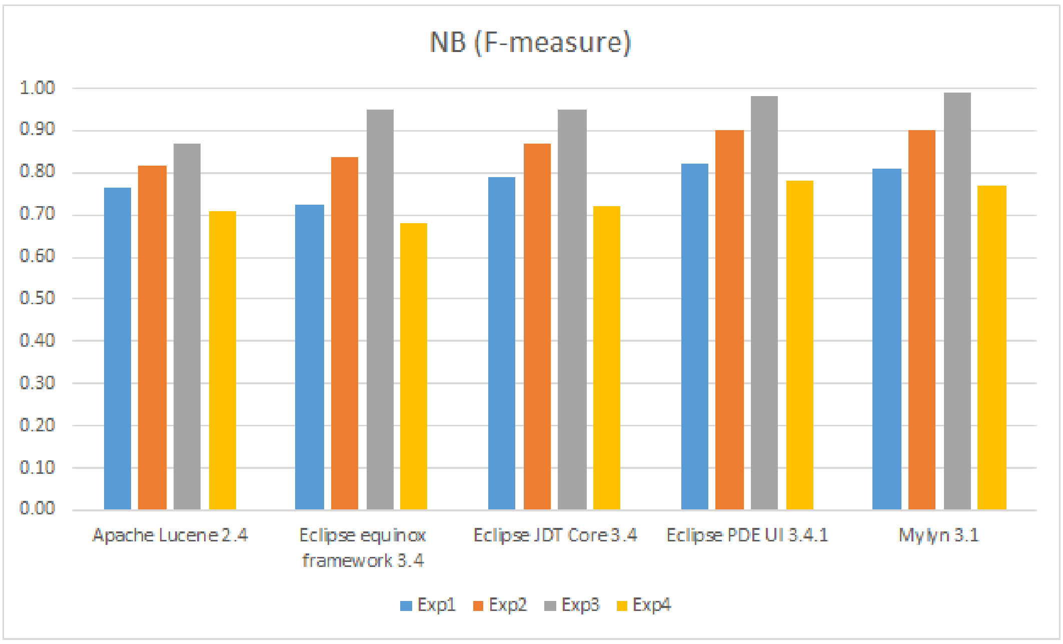

FIGURE 5.26: Difference in F-measure with NB by introducing decomposed Halstead.

Experiment 3 shows that the SVM classifier has the best accuracy of 0.91, while RF and Logistic Regression both gain from the higher granularity at Level 1.

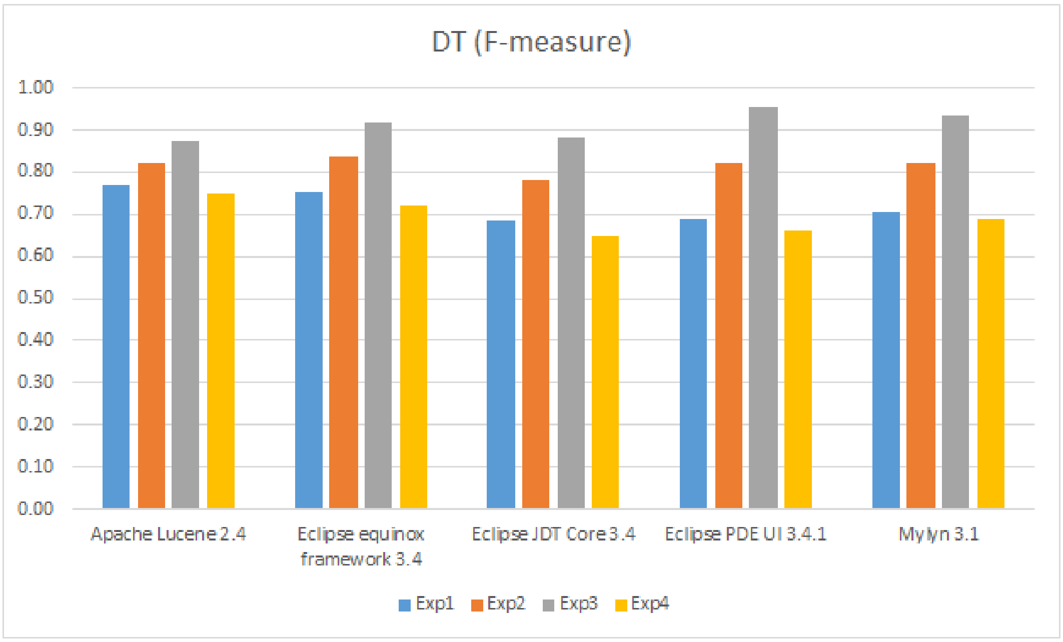

FIGURE 5.27: Difference in F-measure with DT by introducing decomposed Halstead.

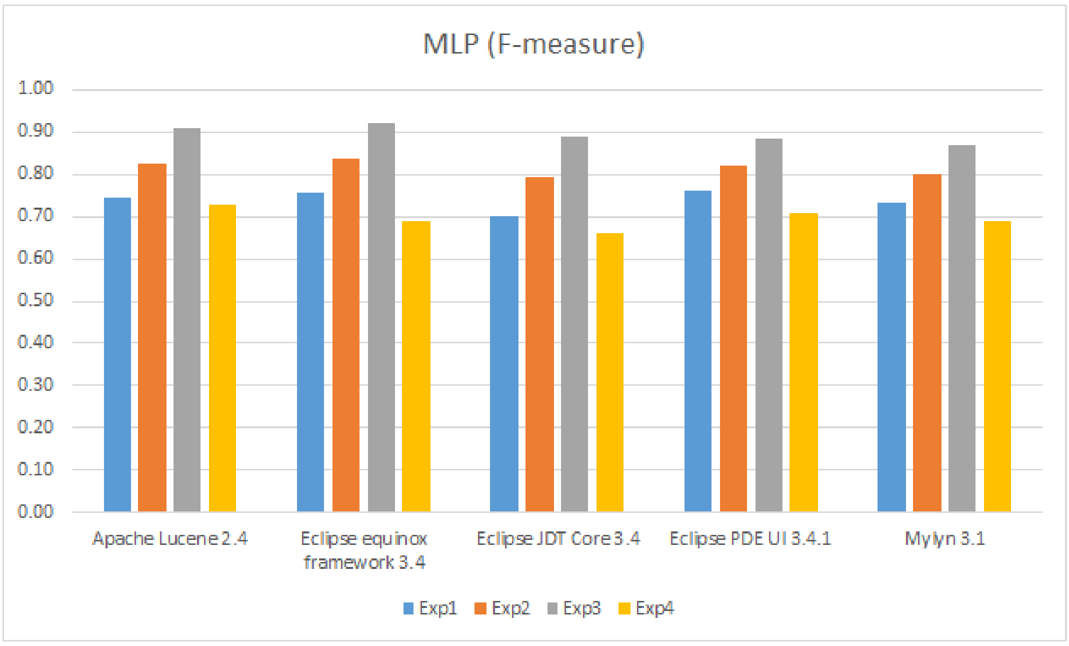

FIGURE 5.28: Difference in F-measure with MLP by introducing decomposed Halstead.

However, as seen in 5.19 to 5.24, the average accuracy for all classifiers drops in Experiment 4, when operators are further decomposed at Level 2, suggesting that excessive feature decomposition increases noise and diminishes generalizability.

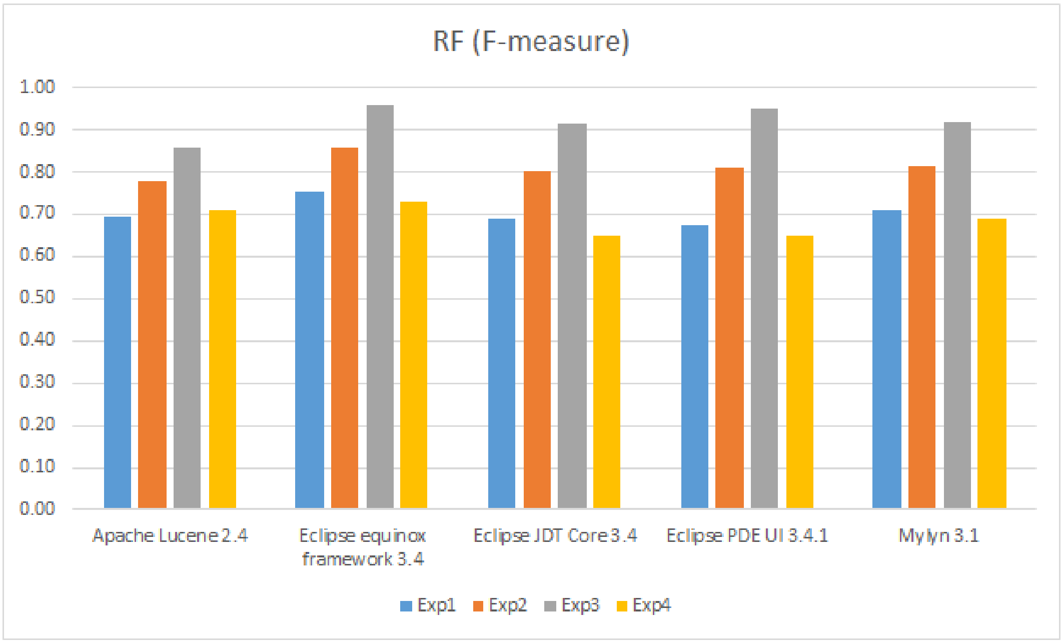

FIGURE 5.29: Difference in F-measure with RF by introducing decomposed Halstead.

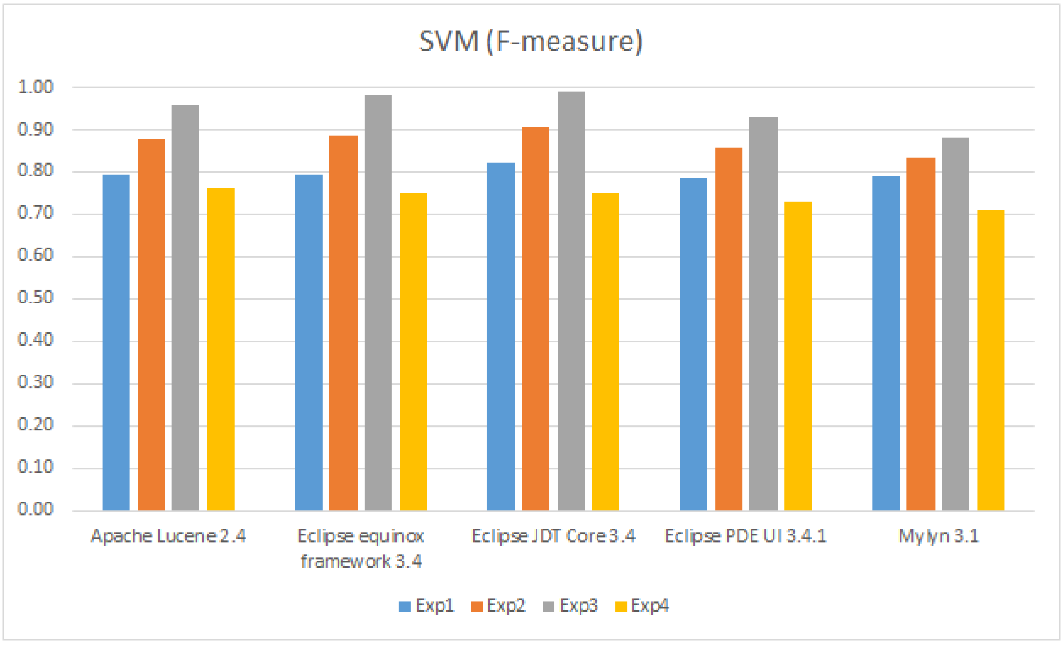

FIGURE 5.30: Difference in F-measure with SVM by introducing decomposed Halstead.

A similar pattern can be seen in the F-measure comparisons in Figures 5.25 to 5.30. In Experiment 3, the F-measure, which weighs recall and precision, peaks for all classifiers, with Random Forest reaching 0.95.

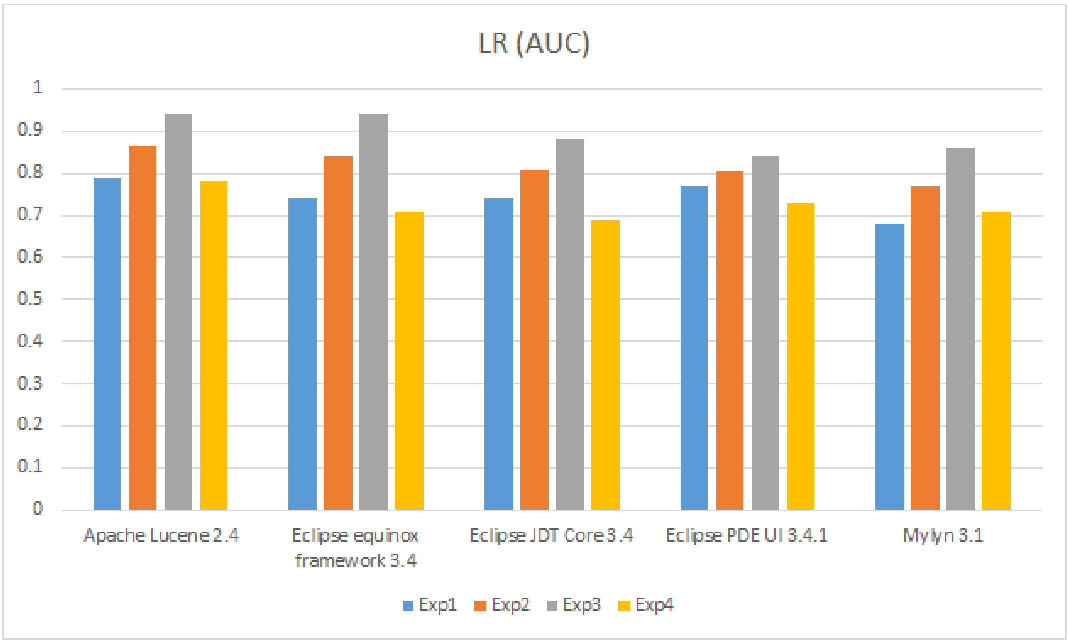

FIGURE 5.31: Difference in AUC with LR by introducing decomposed Halstead.

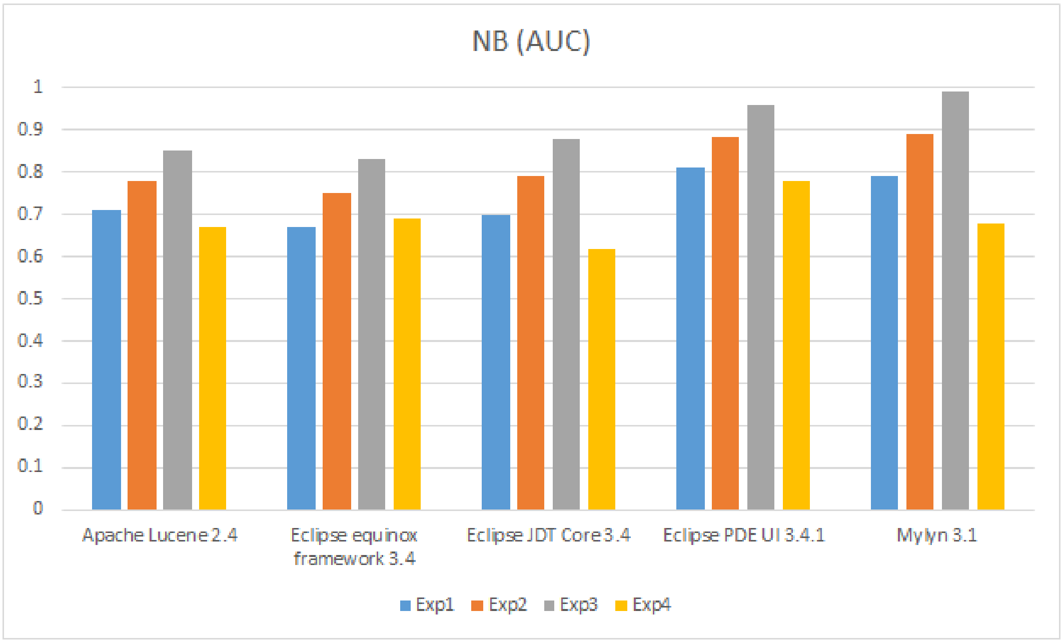

FIGURE 5.32: Difference in AUC with NB by introducing decomposed Halstead.

When both operators and operands are decomposed at Level 1, the RF and Decision Tree models likewise exhibit notable gains.

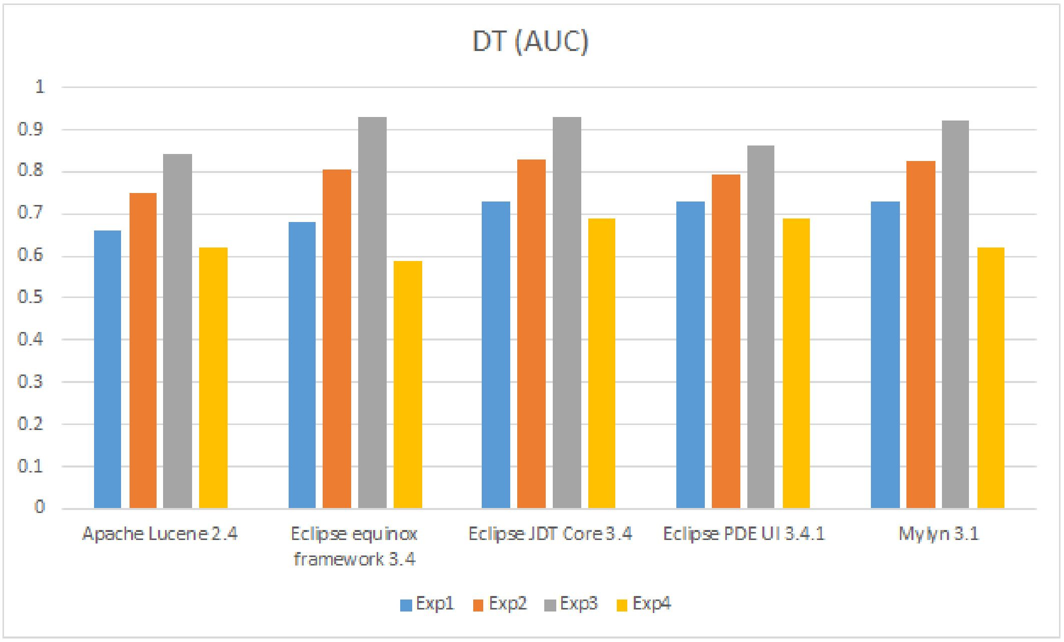

FIGURE 5.33: Difference in AUC with DT by introducing decomposed Halstead.

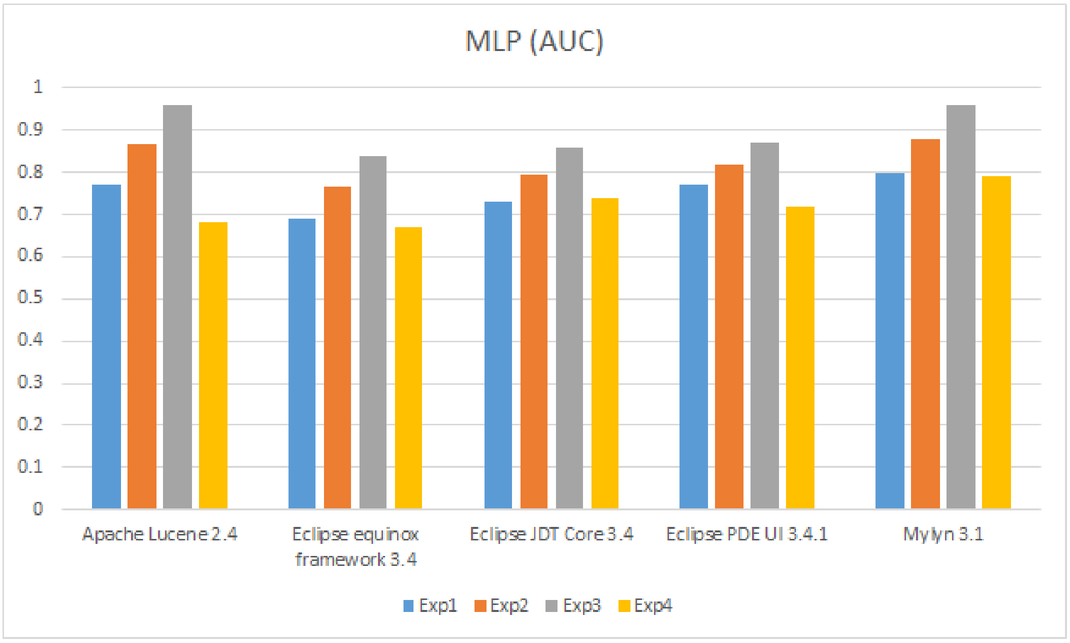

FIGURE 5.34: Difference in AUC with MLP by introducing decomposed Halstead.

But in Experiment 4, the F-measure drops, indicating that deeper decomposition is unable to sustain a balanced prediction, resulting in a higher number of false positives and false negatives. The SVM classifier F-measure decreases when operators are decomposed at Level 2, as shown in Figures 5.25 to 5.30.

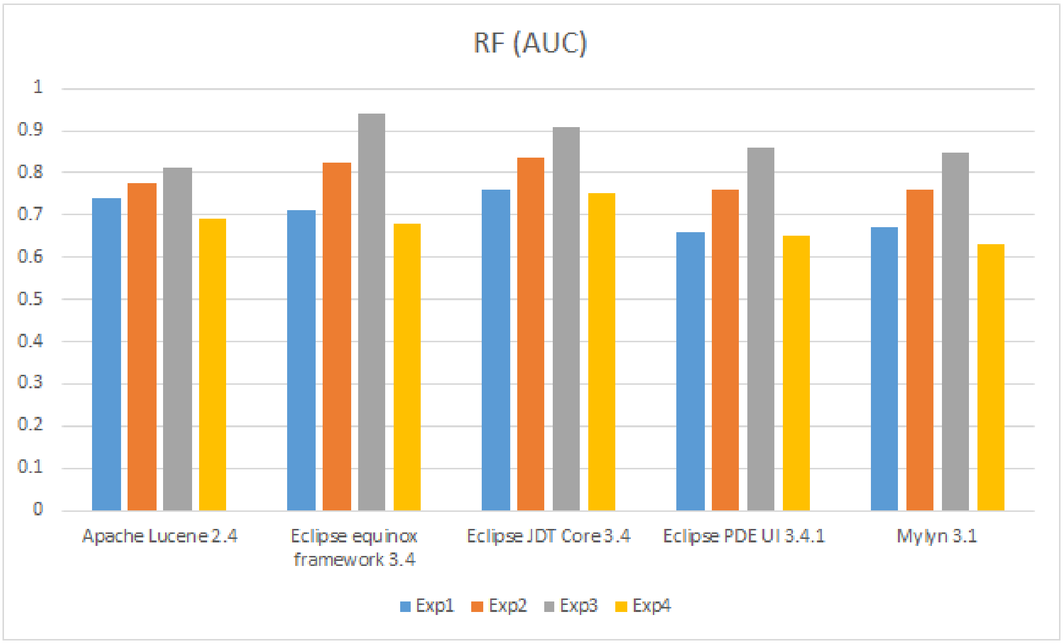

FIGURE 5.35: Difference in AUC with RF by introducing decomposed Halstead.

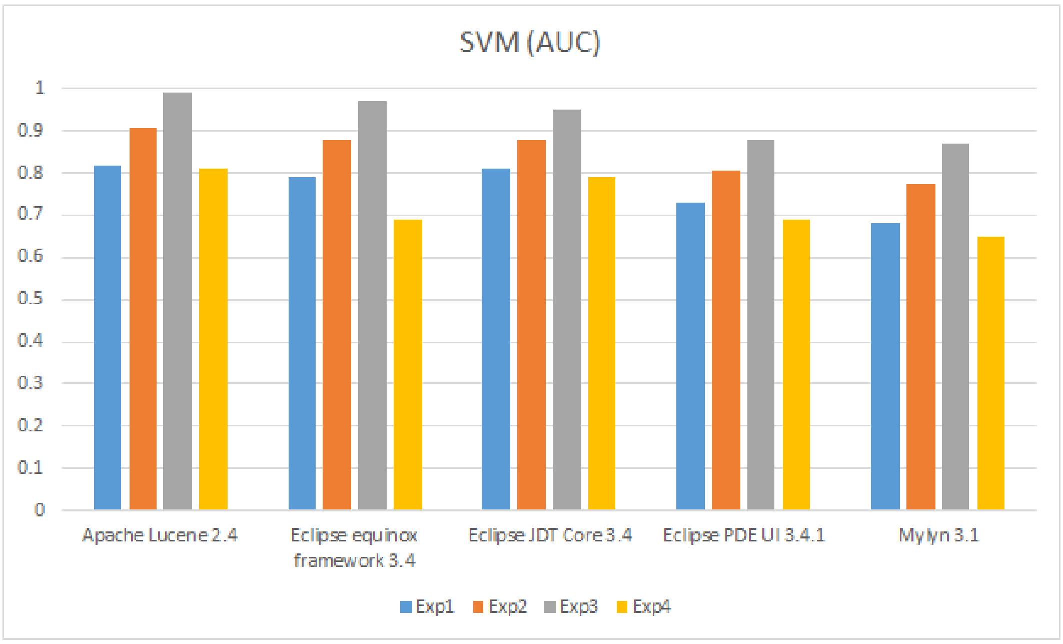

FIGURE 5.36: Difference in AUC with SVM by introducing decomposed Halstead.

The benefit of Level 1 decomposition for differentiating between fault-prone and non-fault-prone modules is confirmed by the AUC results in Figures 5.31 to 5.36. In Experiment 3, SVM has the highest AUC values of 0.98, closely followed by RF and Logistic Regression.

Conversely, Experiment 4 exhibits a discernible loss in AUC, especially for SVM from 0.93 to 0.73 and Logistic Regression from 0.91 to 0.70. These numbers show that while a certain amount of decomposition enhances predictive power, excessive decomposition reduces the feature set's usefulness.

In conclusion, the combined experimental findings show that the best compromise between feature richness and model performance is achieved by decomposition of both Halstead operators and operands at Level 1. While Experiment 4 fall confirms that deeper decomposition adds redundancy and complexity rather than enhancing predictions, Figures 5.19 to 5.36 consistently demonstrate that Experiment 3 performs better than the other configurations across accuracy, F-measure, and AUC measures.

These results support the theory that Level 1 decomposition is the best method for predicting software faults because it captures the pertinent intricacies of software modules without overloading the models.

5.8 Feature Importance Analysis and Impact on Experimental Results

The Random Forest feature ranking score is depicted in Table 5.14, demonstrate how decomposed Halstead operators and operands significantly increase the accuracy of fault prediction.

Features pertaining to decomposed operators and operands were regularly listed among the top contributors, according to the ranking, highlighting their crucial function in differentiating fault-prone modules from non-fault-prone ones.

The better performance seen in Experiment 3, where both operators and operands were decomposed at Level 1, is empirically supported by the higher-ranking scores for these decomposed features. The decomposed features added more significant representations of code complexity to the feature set by capturing fine-grained interactions inside the code.

The classifiers' capacity to identify intricate patterns linked to software flaws was strengthened by this better representation, which raised accuracy, F-measure, and AUC scores.

TABLE 5.14: Feature ranking average score across five selected datasets

Feature	Score
DistinctAOpr	0.89
TotalROpr	0.87
TotalAOpr	0.87
N1	0.83
LoEx	0.78
TotalOthers	0.76
LoB	0.76
iv_g	0.71
LoCoCm	0.7
LoCm	0.7
LoC	0.64
n1	0.61
TotalLOpr	0.61

v_g	0.58
D	0.57
DistinctLOpr	0.57
DistinctROpr	0.56
V	0.55
E	0.54
N	0.54
L	0.50
TotalVariables	0.51
DistinctOthers	0.51
ev_g	0.48
TotalConstants	0.42
DistinctConstants	0.38
N2	0.37
n2	0.32
DistinctVariables	0.30
T	0.28
B	0.22

Conversely, Experiment 1 traditional measurements lower significance scores highlight the drawbacks of employing aggregate-level features. Likewise, Experiment 4 comparatively lower feature ranking lends credence to the conclusion that extra noise is introduced by deeper decomposition at Level 2, which lowers the models' predictive ability.

Decomposed Halstead operators and operands at Level 1 provide the best trade-off between feature granularity and model interpretability, according to the feature ranking score. Their strong significance scores, which are consistent with the performance improvements seen in the earlier experimental results, confirm how well they work to improve software fault prediction.

5.9 Implications of Decomposed Halstead on Classification Model Complexity

To answer our RQ2 we analyses the implications of decomposed Halstead on classification model complexity. The complexity of decision boundaries in classification models is greatly impacted by the decomposed Halstead Software Metrics into smaller, more manageable parts, such as individual operators and operands. In light of the ML models employed in this investigation, these consequences are examined in this section.

5.9.1 Impact on Decision Boundaries Across ML Models

Decision Trees and Random Forest: Decision trees become more complicated and have deeper branches as a result of the decomposition, which increases the number of possible splits. But by averaging several trees, Random Forest lessens this complexity, increasing resilience and lowering the chance of over fitting.

Support Vector Machine (SVM): The hyperplane borders become increasingly complex due to the extra characteristics that come from decomposition, especially when non-linear kernels are used. This may raise the computational requirements but enables the SVM to catch more subtle distinctions.

Multi-Layer Perceptron (MLP): By increasing the number of input neurons and adding more weights and biases, the decomposition can improve pattern recognition; however, it may necessitate more epochs and regularization to prevent over fitting.

5.9.2 Implications of Decomposition on Model Performance

Improved predictive power: The models' capacity to identify minute patterns in the data is improved by the enriched feature set derived from decomposed metrics, which raises classification accuracy throughout the experiments.

Risk of over fitting: Models such as decision trees and MLP may over fit the training data due to the increased dimensionality, particularly for smaller datasets. To solve this problem, methods including regularization, pruning, and dropout were taken into consideration (for MLP, decision trees, and MLP, respectively).

5.9.3 Interpretability vs. Complexity:

Random Forest: Notwithstanding the intricacy, feature importance ratings can draw attention to the most significant aspects, improving the interpretability of the findings.

SVM and MLP: The superior classification performance of these models justifies their usage for some defect prediction tasks, even though their greater feature count may make them more difficult to comprehend.

5.9.4 Computational Considerations

For models like MLP and SVM, the breakdown increased the input size, resulting in longer training periods and higher memory use. However, considering the increase in prediction performance, this trade off was justified.

It is evident that although decision boundaries become more complex, the decomposition process results in more precise and nuanced fault prediction when the decomposition approach is linked to the particular ML models employed in this investigation. The study shows that performance, interpretability, and computing efficiency may all be balanced with the right model selection and optimization strategies.

5.10 Applications of the Study

This study investigation on the decomposed of Halstead base metrics for software defect prediction has a number of practical uses in software engineering and other fields. These applications show how the suggested method can be used in practice to increase software maintainability, efficiency, and dependability.

5.10.1 Software Quality Assurance and Testing Prioritization

Software quality assurance teams can concentrate their testing efforts thanks to the more accurate fault prediction made possible by the suggested decomposed Halstead base metrics. Focusing on modules that are prone to errors, as determined by Level 1 decomposition, makes the testing process more effective overall, saving money and time.

5.10.2 Automated Fault Detection in CI/CD Pipelines

Automated defect detection is essential to contemporary software development methods, especially those that incorporate continuous integration and deployment (CI/CD). More stable and fault-resistant releases can be ensured by improving the capacity to identify flaws early in the development cycle by the incorporation of decomposed Halstead base metrics into fault prediction systems.

5.10.3 Enhanced Static Code Analysis Tools

Static analysis tools can incorporate decomposed Halstead base metrics to give developers more thorough feedback. These tools can provide developers with useful information at the coding stage by detecting particular types of operators and operands linked to increased fault probability, which will lower post-deployment problems.

5.10.4 Resource Allocation in Development and Maintenance

Project managers can more efficiently distribute resources by pinpointing particular parts of the codebase that are more likely to have errors. The capacity to concentrate efforts on high risk modules guarantees the effective use of staff and time, enhancing project management as a whole.

5.10.5 Predictive Maintenance in Safety Critical Systems

High degrees of dependability are necessary in safety-critical fields like financial systems, healthcare, and aviation. The suggested decomposition technique can enhance predictive maintenance models in these areas, enabling businesses to anticipate problems and prevent system breakdowns that could have dire repercussions.

5.10.6 Development of Customizable Fault Prediction Models

This study's methodology can serve as a basis for developing fault prediction models tailored to a particular domain. Organizations can tailor their models to meet their specific requirements by altering the decomposition levels according to the software system's characteristics.

The wide range of uses for the suggested decomposed Halstead base metrics shows how they can influence various facets of software development, including resource planning, predictive maintenance, and quality assurance. This paper adds to the larger field of software engineering by illustrating these useful applications and emphasizes the significance of granular measurements in fault prediction.

5.11 Threats to Validity

The results of our experiment allow us to associate decomposed Halstead base metrics with SFP. Nevertheless, before we could accept the result, we would have to consider possible threats to its validity.

Concerning the size of the projects, sufficient comprehensible project size is taken. The projects of a very large size or very small size were ignored. The reason was the unavailability of either projects' source code or fault information. Therefore, such very large size or small size project may differ in the results reported.

The selected open-source projects are developed in Java, which sufficiently justifies the objective of the experiment and successfully demonstrates the experimental methodology. However, since the decomposed Halstead metric varies in different programming languages. The results may vary when using projects developed in languages other than Java.

Chapter 6

Conclusion and Future Work

Feature decomposition in Halstead base metrics for software fault prediction (SFP) has transformed the landscape of software reliability analysis. Through the meticulous breakdown of traditional Halstead metrics into more nuanced components such as conditionals, decisions, and literals, machine learning algorithms can now extract highly informative features that intricately reflect the complexities of software systems.

The decomposition of Halstead base metrics has enhancements in machine learning algorithms, particularly in the realm of fault prediction. By the decomposed features, models exhibit heightened predictive accuracy, facilitating more reliable assessments of software quality and vulnerability detection. Moreover, the increased granularity of decomposed features enables models to capture subtle patterns and dependencies within the data, further enhancing their predictive power.

Additionally, the interoperability and explainability of decomposed features empower software engineers to glean valuable insights from model predictions, fostering more informed decision-making and ultimately contributing to the advancement of software development practices. In conclusion, this research explores two critical aspects of decomposing Halstead base metrics to improve SFP. RQ1 investigates how Halstead base metrics can be decomposed into different types of operators to enhance fault prediction, while RQ2 examines the overall impact of this decomposition on predictive accuracy.

Two levels of decomposition were explored to refine the feature extraction process. In the first level, traditional decomposition techniques segregated operators and operands within the Halstead metrics to provide a foundational breakdown. The second level focused on categorizing operators into distinct groups, such as assignment, logical, arithmetic, and relational operations, allowing for a more nuanced understanding of the software's behavior.

The experimentation revealed intriguing insights into the impact of different decomposition levels on machine learning accuracy. In certain scenarios, particularly when operators were further decomposed into refined categories, predictive accuracy soared to impressive levels of up to 99%. This suggests that the enhanced granularity of decomposition facilitated the identification of subtle patterns, significantly improving predictive performance. However, in some cases, overly detailed decomposition resulted in a sharp decline in accuracy, highlighting the importance of selecting an optimal decomposition strategy.

This underscores the necessity of carefully balancing the granularity of decomposition to avoid introducing noise that could adversely affect model performance. Thus, while deeper levels of decomposition hold the potential to uncover valuable insights, the research emphasizes the importance of rigorous evaluation and optimization to ensure robust and reliable fault prediction models.

6.1 Future Work

In future work, researchers can extend the decomposition process by focusing on refining the operands category within the decomposed Halstead base metrics. Specifically, exploration into the decomposition of variables into more granular subcategories such as literals, primitives, non-primitives, and non-primitive user-defined types can be pursued. By dissecting the operands into these finer-grained components, researchers can gain deeper insights into the underlying structure and characteristics of software code, thereby enhancing the effectiveness of software fault prediction models. Moreover, the incorporation of additional decomposition layers opens avenues for further improvement by enabling the extraction of more informative features for machine learning algorithms.

Additionally, in order to supplement the improved feature extraction procedure, future research projects may go into the investigation of other machine learning techniques. Through the utilisation of diverse machine learning methodologies such as ensemble methods, probabilistic models, and deep learning, researchers can investigate substitute approaches for software fault prediction that capitalise on the enriched dataset that arises from the advanced decomposition scheme. This all-encompassing method not only improves the models' predictive power but also promotes a greater comprehension of the many relationships and dynamics present in software systems.

Furthermore, the incorporation of explainable AI approaches may provide increased predictability and interpretability, enabling practitioners to understand the fundamental causes of software errors. This may result in better decision-making procedures and the capacity to identify the precise stages of the software development lifecycle that need improvement. The integration of these sophisticated methodologies with the enhanced feature extraction procedure holds promise for augmenting not only the accuracy of fault prediction but also the software development and maintenance procedures in general. Further investigation into these varied directions is expected to yield new insights and patterns that might greatly strengthen software quality assurance protocols.

Ensuring the robustness and generalizability of the produced models will require the deployment of stringent validation approaches, such as cross-validation and real-world testing. Through innovation and the sharing of best practices, collaborative efforts between academics and industry can further accelerate the growth of this sector. All things considered, greater research in this area could greatly progress the field of software fault prediction and aid in the creation of more durable and dependable software engineering procedures. Through the adoption of a comprehensive methodology that blends advanced machine learning techniques with an in-depth comprehension of software systems, researchers can pioneer significant advancements in the precision and efficacy of software fault prediction models.

Bibliography

[1] F. Gurcan, G. G. M. Dalveren, N. E. Cagiltay, D. Roman, and A. Soylu, "Evolution of software testing strategies and trends: Semantic content analysis of software research corpus of the last 40 years," *IEEE Access*, vol. 10, pp. 106 093–106 109, 2022.

[2] V. Garousi and M. V. Mäntylä, "A systematic literature review of literature reviews in software testing," *Information and Software Technology*, vol. 80, pp. 195–216, 2016.

[3] A. Zakari and S. P. Lee, "Simultaneous isolation of software faults for effective fault localization," in *2019 IEEE 15th International Colloquium on Signal Processing & Its Applications (CSPA)*. IEEE, 2019, pp. 16–20.

[4] A. Anand and A. Uddin, "Importance of software testing in the process of software development," *International Journal for Scientfic Research and Development*, vol. 12, no. 6, 2019.

[5] L. Luo, "Software testing techniques," *Institute for Software Research International Carnegie Mellon University Pittsburgh, PA*, vol. 15232, no. 1-19, p. 19, 2001.

[6] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, pp. 1–42, 2014.

[7] A. B. Marques, R. Rodrigues, and T. Conte, "Systematic literature reviews in distributed software development: A tertiary study," in *2012 IEEE Seventh*

International Conference on Global Software Engineering. IEEE, 2012, pp. 134–143.

[8] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," in *12th International Conference on evaluation and Assessment in software engineering (EASE).* BCS Learning & Development, 2008.

[9] S. Nidhra and J. Dondeti, "Black box and white box testing techniques-a literature review," *International Journal of Embedded Systems and Applications (IJESA)*, vol. 2, no. 2, pp. 29–50, 2012.

[10] A. Verma, A. Khatana, and S. Chaudhary, "A comparative study of black box testing and white box testing," *International Journal of Computer Sciences and Engineering*, vol. 5, no. 12, pp. 301–304, 2017.

[11] M. E. Khan and F. Khan, "A comparative study of white box, black box and grey box testing techniques," *International Journal of Advanced Computer Science and Applications*, vol. 3, no. 6, 2012.

[12] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V. Mäntylä, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *2012 7th International Workshop on Automation of Software Test (AST).* IEEE, 2012, pp. 36–42.

[13] M. K. Thota, F. H. Shajin, P. Rajesh *et al.*, "Survey on software defect prediction techniques," *International Journal of Applied Science and Engineering*, vol. 17, no. 4, pp. 331–344, 2020.

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

[15] C. Catal, "Software fault prediction: A literature review and current trends," *Expert systems with applications*, vol. 38, no. 4, pp. 4626–4636, 2011.

[16] G. Abaei and A. Selamat, “A survey on software fault detection based on different prediction approaches,” *Vietnam J. of Computer Science*, vol. 1, no. 2, p. 79–95, may 2014. [Online]. Available: https://doi.org/10.1007/s40595-013-0008-z

[17] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, “Comparing negative binomial and recursive partitioning models for fault prediction,” in *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, ser. PROMISE ’08. New York, NY, USA: ACM, 2008, pp. 3–10.

[18] S. Watanabe, H. Kaiya, and K. Kaijiri, “Adapting a fault prediction model to allow inter languagereuse,” in *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, ser. PROMISE ’08. New York, NY, USA: ACM, 2008, pp. 19–24.

[19] Capgemini, “Sogeti:world quality report 2015–16,” World quality report 2015–16, Tech. Rep., 2015.

[20] S. A. Sherer, “Software fault prediction,” *Journal of Systems and Software*, vol. 29, no. 2, pp. 97–105, 1995.

[21] S. S. Rathore and S. Kumar, “A study on software fault prediction techniques,” *Artificial Intelligence Review*, vol. 51, no. 2, pp. 255–327, Feb 2019.

[22] K. Sandeep and S. R. Santosh, *Software Fault Prediction, A Road Map*. Singapore: Springer Singapore, 2018.

[23] M. W. Morris, B. L. Shaw, and C. D. Ziomek, “Modular & benchtop instrument convergence decreases test costs and increases productivity,” in *2007 IEEE Autotestcon*. IEEE, 2007, pp. 284–290.

[24] N. Seliya and T. M. Khoshgoftaar, “Software quality estimation with limited fault data: a semi-supervised learning perspective,” *Software Quality Journal*, vol. 15, no. 3, pp. 327–344, 2007.

[25] R. Marinescu, “Assessing technical debt by identifying design flaws in software systems,” *IBM Journal of Research and Development*, vol. 56, no. 5, pp. 9–1, 2012.

[26] I. H. Sarker, “Machine learning: Algorithms, real-world applications and research directions,” *SN computer science*, vol. 2, no. 3, p. 160, 2021.

[27] N. Burkart and M. F. Huber, “A survey on the explainability of supervised machine learning,” *Journal of Artificial Intelligence Research*, vol. 70, pp. 245–317, 2021.

[28] F. Hahne, W. Huber, R. Gentleman, S. Falcon, R. Gentleman, and V. Carey, “Unsupervised machine learning,” *Bioconductor case studies*, pp. 137–157, 2008.

[29] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[30] J. M. Chambers and T. J. Hastie, “Statistical models,” in *Statistical models in S*. Routledge, 2017, pp. 13–44.

[31] G. K. Uyanık and N. Güler, “A study on multiple linear regression analysis,” *Procedia-Social and Behavioral Sciences*, vol. 106, pp. 234–240, 2013.

[32] S. Sperandei, “Understanding logistic regression analysis,” *Biochemia medica*, vol. 24, no. 1, pp. 12–18, 2014.

[33] S. Abd ElHafeez, G. D’Arrigo, D. Leonardis, M. Fusaro, G. Tripepi, and S. Roumeliotis, “Methods to analyze time-to-event data: the cox regression analysis,” *Oxidative medicine and cellular longevity*, vol. 2021, pp. 1–6, 2021.

[34] A. Omer, S. S. Rathore, and S. Kumar, “Me-sfp: A mixture-of-experts-based approach for software fault prediction,” *IEEE Transactions on Reliability*, 2023.

[35] T. M. Khoshgoftaar, N. Seliya, and N. Sundaresh, “An empirical study of predicting software faults with case-based reasoning,” *Software Quality Journal*, vol. 14, pp. 85–111, 2006.

[36] G. Czibula, Z. Marian, and I. G. Czibula, “Software defect prediction using relational association rule mining,” *Information Sciences*, vol. 264, pp. 260–278, 2014.

[37] Q. Zhao and S. S. Bhowmick, “Association rule mining: A survey,” *Nanyang Technological University, Singapore*, vol. 135, 2003.

[38] C. Catal and B. Diri, “A systematic review of software fault prediction studies,” *Expert Syst. Appl.*, vol. 36, no. 4, pp. 7346–7354, May 2009.

[39] S. S. Rathore and S. Kumar, “A decision tree logic based recommendation system to select software fault prediction techniques,” *Computing*, vol. 99, no. 3, pp. 255–285, 2017.

[40] S. K. Pandey, R. B. Mishra, and A. K. Tripathi, “Machine learning based methods for software fault prediction: A survey,” *Expert Systems with Applications*, vol. 172, p. 114595, 2021.

[41] E. E. Mills, *Software metrics.* Software Engineering Institute, 1988.

[42] H. F. Li and W. K. Cheung, “An empirical study of software metrics,” *IEEE Transactions on Software Engineering*, no. 6, pp. 697–708, 1987.

[43] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya, “Choosing software metrics for defect prediction: an investigation on feature selection techniques,” *Software: Practice and Experience*, vol. 41, no. 5, pp. 579–606, 2011.

[44] M. Jureczko, “Significance of different software metrics in defect prediction,” *Software Engineering: An International Journal*, vol. 1, no. 1, pp. 86–95, 2011.

[45] H. Wang, T. M. Khoshgoftaar, and N. Seliya, “How many software metrics should be selected for defect prediction?” in *Twenty-Fourth International FLAIRS Conference*, 2011.

[46] S. SENTHILNATHAN and V. SANGEETHA, “Performance analysis of reliability, quality and metric analysis fault prediction assessment for software testing.” *International Journal of Advanced Research in Computer Science*, vol. 8, no. 9, 2017.

[47] R. A. Paul, T. L. Kunii, Y. Shinagawa, and M. F. Khan, “Software metrics knowledge and databases for project management,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 1, pp. 255–264, 1999.

[48] M. Bhardwaj and A. Rana, "Key software metrics and its impact on each other for software development projects," *ACM SIGSOFT Software Engineering Notes*, vol. 41, no. 1, pp. 1–4, 2016.

[49] A.-J. Molnar, A. Neamţu, and S. Motogna, "Evaluation of software product quality metrics," in *Evaluation of Novel Approaches to Software Engineering: 14th International Conference, ENASE 2019, Heraklion, Crete, Greece, May 4–5, 2019, Revised Selected Papers 14*. Springer, 2020, pp. 163–187.

[50] R. Malhotra and A. Bansal, "Predicting change using software metrics: A review," in *2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)(Trends and Future Directions)*. IEEE, 2015, pp. 1–6.

[51] W. Li, "Software product metrics," *IEEE Potentials*, vol. 18, no. 5, pp. 24–27, 1999.

[52] S. H. Kan, J. Parrish, and D. Manlove, "In-process metrics for software testing," *IBM Systems Journal*, vol. 40, no. 1, pp. 220–241, 2001.

[53] T. L. Woodings and G. A. Bundell, "A framework for software project metrics," in *Proc. 12th European Conference on Software Control and Metrics (ESCOM'01)*, 2001.

[54] M. Shepperd, D. Bowes, and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 603–616, 2014.

[55] M. E. Khan and F. Khan, "Importance of software testing in software development life cycle," *International Journal of Computer Science Issues (IJCSI)*, vol. 11, no. 2, p. 120, 2014.

[56] V. Garousi, M. Felderer, M. Kuhrmann, K. Herkiloğlu, and S. Eldh, "Exploring the industry's challenges in software testing: An empirical study," *Journal of Software: Evolution and Process*, vol. 32, no. 8, p. e2251, 2020.

[57] J. Pachouly, S. Ahirrao, K. Kotecha, G. Selvachandran, and A. Abraham, "A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools," *Engineering Applications of Artificial Intelligence*, vol. 111, p. 104773, 2022.

[58] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, 2011.

[59] K. Bhandari, K. Kumar, and A. L. Sangal, "Data quality issues in software fault prediction: a systematic literature review," *Artificial Intelligence Review*, vol. 56, no. 8, pp. 7839–7908, 2023.

[60] D. Radjenovi, M. Heri, R. Torkar, and A. ivkovi, "Software fault prediction metrics: A systematic literature review," *Information and software technology*, vol. 55, no. 8, pp. 1397–1418, 2013.

[61] I. Batool and T. A. Khan, "Software fault prediction using data mining, machine learning and deep learning techniques: A systematic literature review," *Computers and Electrical Engineering*, vol. 100, p. 107886, 2022.

[62] N.-H. Chiu, "Combining techniques for software quality classification: An integrated decision network approach," *Expert Systems with Applications*, vol. 38, no. 4, pp. 4618–4625, 2011.

[63] K. Dejaeger, T. Verbraken, and B. Baesens, "Toward comprehensible software fault prediction models using bayesian network classifiers," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 237–257, Feb 2013.

[64] Ö. F. Arar and K. Ayan, "Software defect prediction using cost-sensitive neural network," *Applied Soft Computing*, vol. 33, pp. 263–277, 2015.

[65] R. C. G. Dhanajayan and S. A. Pillai, "Slmbc: spiral life cycle model-based bayesian classification technique for efficient software fault prediction and classification," *Soft Computing*, vol. 21, no. 2, pp. 403–415, 2017.

[66] G. P. Bhandari and R. Gupta, “Machine learning based software fault prediction utilizing source code metrics,” in *2018 IEEE 3rd International Conference on Computing, Communication and Security (ICCCS).* IEEE, 2018, pp. 40–45.

[67] T. Shippey, D. Bowes, and T. Hall, “Automatically identifying code features for software defect prediction: Using ast n-grams,” *Information and Software Technology*, vol. 106, pp. 142–160, 2019.

[68] M. Cetiner and O. K. Sahingoz, “A comparative analysis for machine learning based software defect prediction systems,” pp. 1–7, 2020.

[69] V. Kumar, Kumar, and D. Mohapatra, “Software fault prediction using lssvm with different kernel functions,” *Arabian Journal for Science and Engineering*, vol. 46, 04 2021.

[70] C. F. Caiafa, J. Solé-Casals, P. Marti-Puig, S. Zhe, and T. Tanaka, “Decomposition methods for machine learning with small, incomplete or noisy datasets,” *Applied Sciences*, vol. 10, no. 23, p. 8481, 2020.

[71] O. Maimon and L. Rokach, “Improving supervised learning by feature decomposition,” in *International Symposium on Foundations of Information and Knowledge Systems.* Springer, 2002, pp. 178–196.

[72] T. De Quadros, A. E. Lazzaretti, and F. K. Schneider, “A movement decomposition and machine learning-based fall detection system using wrist wearable device,” *IEEE Sensors Journal*, vol. 18, no. 12, pp. 5082–5089, 2018.

[73] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, “A systematic literature review on fault prediction performance in software engineering,” *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, 2012.

[74] R. Malhotra, “A systematic review of machine learning techniques for software fault prediction,” *Applied Soft Computing*, vol. 27, no. C, pp. 504–518, Feb. 2015.

[75] R. S. Wahono, “A systematic literature review of software defect prediction,” *Journal of Software Engineering*, vol. 1, no. 1, pp. 1–16, 2015.

[76] M. Caulo, “A taxonomy of metrics for software fault prediction,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1144–1147.

[77] C. Catal, U. Sevim, and B. Diri, “Software fault prediction of unlabeled program modules,” in *Proceedings of the world congress on engineering*, vol. 1, 2009, pp. 1–3.

[78] I. Batool and T. A. Khan, “Software fault prediction using deep learning techniques,” *Software Quality Journal*, vol. 31, no. 4, pp. 1241–1280, 2023.

[79] G. Singh, D. Singh, and V. Singh, “A study of software metrics,” *IJCEM International Journal of Computational Engineering & Management*, vol. 11, no. 2011, pp. 22–27, 2011.

[80] M. R. Ahmed, M. A. Ali, N. Ahmed, M. F. B. Zamal, and F. J. M. Shamrat, “The impact of software fault prediction in real-world application: an automated approach for software engineering,” in *Proceedings of 2020 the 6th international conference on computing and data engineering*, 2020, pp. 247–251.

[81] M. J. Hernández-Molinos, A. J. Sánchez-García, R. E. Barrientos-Martínez, J. C. Pérez-Arriaga, and J. O. Ocharán-Hernández, “Software defect prediction with bayesian approaches,” *Mathematics*, vol. 11, no. 11, p. 2524, 2023.

[82] S. Haldar and L. F. Capretz, “Interpretable software defect prediction from project effort and static code metrics,” *Computers*, vol. 13, no. 2, p. 52, 2024.

[83] M. Nilsson, “A comparative case study on tools for internal software quality measures,” 2019.

[84] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977.

[85] H. Maurice H, "Natural laws controlling algorithm structure?" *ACM Sigplan Notices*, vol. 7, no. 2, pp. 19–26, 1972.

[86] L. Love and A. Bowman, "An independent test of the theory of software physics," *ACM Sigplan Notices*, vol. 11, no. 11, pp. 42–49, 1976.

[87] A. B. Fitzsimmons, "Relating the presence of software errors to the theory of software science," in *Proc. 11th Hawaii International Conference on Systems Sciences*, vol. 40, 1978, p. 46.

[88] S. M. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Transactions on Software Engineering*, vol. 7, no. 5, pp. 510–518, Sep. 1981.

[89] Z. Tóth, "New datasets for bug prediction and a method for measuring maintainability of legacy software systems," Ph.D. dissertation, University of Szeged, 2019.

[90] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010.

[91] N. Govil, "Applying halstead software science on different programming languages for analyzing software complexity," in *2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184)*. IEEE, 2020, pp. 939–943.

[92] P. Academy, *Python Programming for Beginners, Python Workbook.* FAF PUBLISHING Limited, 2020.

[93] L. Pratt, *PHP: Advanced Guide to Learn the Realms of PHP Programming.* Independently Published, 2021.

[94] N. Feroz, *A Step by Step Guide for Beginners.* Amazon Digital Services LLC - KDP Print US, 2019.

[95] J. Gustedt, *Modern C.* Manning, 2019.

[96] I. Gvero, “Core java volume i: Fundamentals, by cay s. horstmann and gary cornell,” *ACM Sigsoft Software Engineering Notes*, vol. 38, no. 3, pp. 33–33, 2013.

[97] Z. Tóth, P. Gyimesi, and R. Ferenc, “A public bug database of github projects and its application in bug prediction,” in *International Conference on Computational Science and Its Applications*, 2016, pp. 625–638.

[98] S. B. Kotsiantis, I. Zaharakis, P. Pintelas *et al.*, “Supervised machine learning: A review of classification techniques,” *Emerging artificial intelligence applications in computer engineering*, vol. 160, no. 1, pp. 3–24, 2007.