PhD Thesis

BHAWNA JAIN

2K21/PHDCS/503

2026

AN EMPIRICAL STUDY ON ANTIPATTERNS PREDICTABILITY USING MACHINE LEARNING TECHNIQUES FOR ENHANCING SOFTWARE MAINTENANCE

A Thesis
Submitted in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY
in
COMPUTER SCIENCE

by

BHAWNA JAIN
(2K21/PHDCS/503)

Under the supervision of

Prof. Ruchika Malhotra

Professor and Head of Department

Delhi Technological University, Delhi, India

and

Dr. Marouane Kessentini

Dean, College of Computing

Grand Valley State University, Michigan, USA

Department of Software Engineering

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi 110042

October 2026

Copyright © **October 2026**
Delhi Technological University, Shahbad Daulatpur,
Main Bawana Road, Delhi 110042
All rights reserved

ACKNOWLEDGEMENT

Karpuragauram Karunavataram, Sansarasaram Bhujagendraharam

Sadavasantam Hridayaravinde, Bhavam Bhavani Sahitam Namami

At the core of my academic odyssey, I express my heartfelt gratitude to the divine presence for being the unwavering source of strength, resilience, and guidance. In moments of challenge and triumph alike, the divine hand has provided the fortitude needed to navigate the complexities of this academic journey. The belief that there is a higher purpose and a guiding force has been a constant beacon, offering solace and inspiration during the trials and triumphs of my research endeavors.

With a profound sense of gratefulness, I would like to express my deepest gratitude to my supervisor, **Prof. Ruchika Malhotra**, Professor and Head of the Department of Software Engineering, Delhi Technological University (DTU), Delhi, India. Her scholarly guidance, constructive feedback, and unwavering support have been instrumental in shaping this thesis. I am especially grateful for her availability during critical phases of the work and life, and for the many thoughtful discussions that helped refine both the methodology and the presentation of my findings and shaping me to be a better person. Her commitment to academic excellence and her encouragement to pursue rigorous research have left a lasting impact on my development as a researcher. I consider myself very fortunate to have worked under her supervision.

I also extend my sincere thanks to my co-supervisor, **Dr. Marouane Kessentini**, Dean of the College of Computing at Grand Valley State University, Allendale, Michigan, USA. His expertise in software engineering and his keen insights into problem formulation greatly enriched this thesis. His pragmatic approach and constructive challenges pushed me to think more critically and comprehensively. I am deeply thankful for his mentorship and for the professional standards he exemplified.

I extend my heartfelt gratitude to Delhi Technological University for selecting me as a doctoral candidate and providing an environment conducive to academic growth. I am deeply appreciative of **Prof. Prateek Sharma**, Vice-Chancellor of Delhi Technological University, Delhi, India, for his encouragement and for fostering a culture that empowers young researchers to strive for excellence.

I express my warm love and deepest gratitude to my parents, **Sh. Jai Kishan Jain** and **Smt. Raj Bala Jain** whose sacrifices and values have shaped who I am today. Without their support, appreciation, and driving force, it would never have been possible for me to achieve anything in my life. To my sister, Anshu Jain, and my nieces, Ananya and Hitanshi, your unconditional love, unwavering support, and comforting presence have been my quiet strength. I praise my husband, Jatin Jain, my extended in-laws family, and my niece Krisha Jain, for their unending support and for always cheering me on.

I would also like to thank my friends, whose companionship and encouragement brought balance and joy to the challenging phases of this journey. The shared memories and laughter made this experience truly enriching. I extend sincere thanks to my colleagues, peers, students and the faculty and staff of Delhi Technological University and Indira Gandhi Delhi Technical University for Women (IGDTUW). Their cooperation, academic camaraderie, and openness in sharing knowledge helped create a nurturing and intellectually stimulating environment for my research.

Finally, I would like to acknowledge everyone who, in one way or another, contributed to the successful completion of this work. Your encouragement and support, whether big or small, have helped me reach this significant milestone in my life. Thank you all.

This thesis is the culmination of not just my efforts but the collective support, love, and encouragement of all these remarkable individuals. I forever remain grateful for the lasting impact they have had on my academic, professional, and personal life.

Date: October 2026 BHAWNA JAIN



DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Shahbad Daulatpur, Main Bawana Road, Delhi-42

CANDIDATE'S DECLARATION

I, **BHAWNA JAIN**, **2K21/PHDCS/503**, hereby certify that the work which is being presented in the thesis entitled "**An Empirical Study on Antipatterns Predictability using Machine Learning Techniques for Enhancing Software Maintenance**," in partial fulfillment of the requirements for the award of the degree of **Doctor of Philosophy** in **Computer Science** discipline in the **Department of Software Engineering**, Delhi Technological University is an authenticate record of my own work carried out during the period from **January 06, 2022** to **October 14, 2026** under the supervision of **Prof. Ruchika Malhotra**, Department of Software Engineering, Delhi Technological University, Delhi, India, and **Dr. Marouane Kessentini**, College of Computing, Grand Valley State University, Michigan, USA. The matter presented in the thesis has not been submitted by me for the award of any other degree of this or any other Institute.

(Candidate's Signature)

This is to certify that the student has incorporated all the corrections suggested by the examiners in the thesis and the statement made by the candidate is correct to the best of our knowledge.

kessentini

Signature of Supervisor(s) **Signature of External Examiner**



DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi-42

CERTIFICATE BY THE SUPERVISOR(S)

Certified that **BHAWNA JAIN (2K21/PHDCS/503)** has carried out her research work presented in this thesis entitled **"An Empirical Study on Antipatterns Predictability using Machine Learning Techniques for Enhancing Software Maintenance"** for the award of **Doctor of Philosophy** in **Computer Science** from Department of Software Engineering, Delhi Technological University, Delhi, under our supervision. The thesis embodies results of original work, and studies are carried out by the student herself, and the contents of the thesis do not form the basis for the award of any other degree to the candidate or to anybody else from this or any other University/Institution.

kessentini

Prof. Ruchika Malhotra
Professor and HoD
Department of Software Engineering
Delhi Technological University
Delhi, India
Date: October 2026

Dr. Marouane Kessentini
Dean
College of Computing
Grand Valley State University
Grand Rapids, Michigan, USA
Date: October 2026

"This thesis is dedicated to my beloved father Sh. Jai Kishan Jain, who always dreamed of seeing me as Dr. Bhawna Jain,
and whose silent strength continues to guide me.

And to my mother, Smt. Raj Bala Jain,
who is my friend, family, and everything in between -
her unwavering support and unconditional love
have been my greatest source of strength throughout this journey.

Special thanks to Khatu Shyam Ji"

Abstract

Software development often suffers from poor design practices, such as code duplication, hardcoding, and overly complex structures. These practices give rise to *code smells or antipatterns*, which are subtle design issues that do not directly affect functionality but gradually degrade maintainability, flexibility, and readability. Traditional detection methods, including manual reviews and metric-based tools, have proven limited by subjectivity, inconsistency, and a lack of adaptability to evolving software paradigms.

To overcome the limitations of manual and semi-automatic approaches, researchers developed code smell detectors that utilize various information sources. While these detectors achieved high accuracy, they faced three significant limitations: (1) developers' subjective perceptions of code smells, (2) inconsistencies among different detectors, and (3) challenges in determining appropriate detection thresholds, limiting their real-world use. This prompted a shift toward growing Machine Learning (ML) field, with Deep Learning (DL) emerging as a powerful alternative. However, despite encouraging results, there remains a significant gap in understanding the full capabilities of DL-based techniques and identifying areas where further enhancement is needed. This thesis explores the use of DL to improve the detection and understanding of software code smells.

To begin with, a systematic literature review of 35 key studies selected from a larger pool of 8,739 research papers was conducted. The review identifies common code smells and outlines the trends in applying DL models for automated detection. It also highlights open challenges such as class imbalance in datasets, rigid metric thresholds, and a lack of research in the context of quantum software.

To address class imbalance, a new data resampling technique called *DIR-SMOTE* is proposed, leading to measurable improvements in model performance. Building on this,

the thesis explores the effectiveness of ensemble methods, both homogeneous and heterogeneous hybrid models, to boost the accuracy of DL techniques in code smell detection. Additionally, a hybrid GNN-based approach was proposed, which constructs a heterogeneous code graph to capture both structural and semantic relationships within source code. By leveraging Graph Convolutional Networks (GCNs) and Graph Attention Networks (GATs), this method enables deeper and more meaningful feature learning. These studies were conducted using datasets derived from classical system programs.

Another significant observation from this research was the lack of exploration in detecting code smells in quantum programs. Due to the limited availability of quantum datasets containing code smells, the thesis defines a new set of structural and semantic metrics, *QUAMET*, specifically designed for quantum programs. A labeled dataset is manually curated, enabling the application of DL models for smell classification in this emerging domain. In addition, Python-based real-world quantum codebases are used to create textual datasets, helping bridge the gap between theoretical research and practical implementation. The thesis also leverages evolutionary algorithms for feature selection by introducing *mQIEA*, ensuring that only the most relevant and predictive features contribute to model performance. This approach not only improves accuracy but also simplifies interpretability. Lastly, existing tools for quantum code smell detection are evaluated in terms of accuracy and user experience, offering a clearer picture of the current capabilities and limitations in this space.

The contributions of this work hold strong implications for both classical and quantum software engineering. By improving detection techniques and broadening their applicability, this research supports the development of cleaner, more maintainable, and more reliable software. Its outcomes can significantly reduce technical debt and improve developer productivity, while also paving the way for better quality assurance in emerging quantum computing environments, thus contributing to the broader goal of building trustworthy and future-proof digital systems.

Contents

LIST OF TABLES xiv
LIST OF FIGURES xvii
LIST OF PUBLICATIONS xx
LIST OF ABBREVIATIONS xxii

1 Introduction **1**
1.1 Software Maintenance 2
1.1.1 Types of Software Maintenance 2
1.1.2 Software Maintainability and its Importance 3
1.2 Empirical Study 4
1.2.1 Steps in Empirical Study 4
1.2.2 Predictive Modeling for Software Maintainability 5
1.2.3 Issues in Predictive Modelling 6
1.3 Design Patterns 7
1.3.1 SOLID Principles 7
1.3.2 Violation of SOLID Principles 9
1.4 Antipatterns 10
1.4.1 Design Antipatterns 10
1.4.2 Code Smells 11
1.5 Literature review 12
1.5.1 Classical Code Smell Detection 13
1.5.2 Quantum Code Smell Detection 16
1.6 Objective of the Thesis 17
1.6.1 Motivation and Scope 17
1.6.2 Research Gaps 20
1.6.3 Problem Statement 21
1.6.4 Vision 21
1.6.5 Focus 21
1.6.6 Goals 22

1.7 Overview of the Work . . . 25
1.8 Organization of Thesis . . . 27

2 Research Methodology **32**
2.1 Research Process . . . 32
2.2 Defining Research Problem . . . 34
2.3 Literature Survey . . . 35
2.3.1 Steps to Conduct Literature Survey . . . 35
2.4 Defining Variables . . . 37
2.4.1 Independent Variables . . . 38
2.4.2 Dependent Variables . . . 42
2.5 Data Analysis Methods . . . 53
2.5.1 Machine Learning Classifiers . . . 53
2.5.2 Deep Learning Classifiers . . . 64
2.5.3 Evolutionary Algorithms . . . 72
2.5.4 AutoML . . . 77
2.6 Experimental Design Framework . . . 79
2.6.1 Empirical Data Collection . . . 79
2.6.2 Software Systems used for Data Collection . . . 80
2.6.3 Data Preprocessing . . . 83
2.6.4 Data Balancing . . . 84
2.6.5 Prediction Model and Validation . . . 84
2.6.6 Performance Measures . . . 86
2.6.7 Statistical Analysis . . . 88

3 Systematic Literature Review **91**
3.1 Background . . . 91
3.2 Overview of the SLR Process . . . 94
3.2.1 Planning . . . 94
3.2.2 Research Questions . . . 94
3.2.3 Search Strategy . . . 96
3.2.4 Article Selection Process . . . 98
3.3 Results of the Review . . . 104
3.3.1 Results Specific to RQ1 . . . 106
3.3.2 Results Specific to RQ2 . . . 107
3.3.3 Results Specific to RQ3 . . . 116
3.3.4 Results Specific to RQ4 . . . 118

3.4 Discussions . . . 122
3.4.1 Outcomes of Literature Review . . . 123
3.4.2 Research Gaps . . . 124

4 DIR-SMOTE for Imbalanced Code Smell Detection 126
4.1 Introduction . . . 126
4.2 Methodology . . . 129
4.2.1 Datasets Used . . . 130
4.2.2 Feature Representation and Input Construction . . . 131
4.2.3 Preprocessing . . . 132
4.2.4 Training Base Learner Classifier Model . . . 132
4.2.5 Model Validation . . . 133
4.3 Proposed DIR-SMOTE Framework . . . 134
4.3.1 SHAP Value Computation . . . 135
4.3.2 Density Weighting . . . 135
4.3.3 Sample Allocation . . . 136
4.3.4 SHAP-Guided Feature Perturbation . . . 137
4.3.5 Dataset Augmentation . . . 138
4.4 Results . . . 138
4.4.1 RQ1: Which base classifier yields the highest performance for the initial model training across imbalanced datasets? . . . 138
4.4.2 RQ2: How effectively can SHAP-Based Explanations identify the most influential features for synthetic data generation? . . . 141
4.4.3 RQ3: Which density estimation technique provides the most reliable weighting for minority class samples? . . . 143
4.4.4 RQ4: How does DIR-SMOTE perform against existing oversampling methods in improving classification on imbalanced datasets? . . . 145
4.4.5 RQ5: How does DIR-SMOTE perform across different learning models? . . . 151
4.4.6 Error Analysis and Failure Cases . . . 154
4.4.7 Statistical Significance and Effect Size Analysis . . . 155
4.5 Running Example of DIR-SMOTE . . . 156
4.6 Discussion . . . 158

5 Advanced Deep Learning and Graph-Based Approaches 160
5.1 Empirical Study on Feature Extraction using Ensemble Deep Learning for Code Smell Detection . . . 160

5.2 Considered Framework . . . 161
5.2.1 Empirical Study . . . 163
5.2.2 Results of Ensemble Models . . . 165
5.2.3 Discussion . . . 168
5.3 SmellGraph . . . 169
5.3.1 Methodology . . . 170
5.4 Empirical Study . . . 173
5.4.1 Dataset and Pre-Processing . . . 173
5.4.2 Baselines . . . 173
5.4.3 Validation and Evaluation . . . 174
5.5 Results of SmellGraph . . . 174
5.5.1 RQ1: Effectiveness . . . 174
5.5.2 RQ2: Ablation . . . 176
5.5.3 RQ3: Cross-Project Generalisation . . . 176
5.5.4 RQ4: Attribution Maps . . . 177
5.5.5 Discussion . . . 177

6 QUAMET: Structural and Semantic Metrics for Quantum Code Smell 178
6.1 Introduction . . . 179
6.2 Background . . . 183
6.3 Methodology . . . 184
6.3.1 Dataset and Preprocessing . . . 184
6.3.2 Structural Metric Analysis . . . 186
6.3.3 Semantic Metric Analysis . . . 188
6.3.4 Quantum Specific Code Smells . . . 189
6.3.5 Selected Machine Learning Classifiers . . . 195
6.3.6 Statistical Tests . . . 197
6.4 Running Example: Bernstein-Vazirani Circuit . . . 198
6.5 Results Specific to RQs . . . 199
6.5.1 Setup Requirements . . . 199
6.5.2 RQ1: What are the researchers' views on quantum-specific code smells? . . . 200
6.5.3 RQ2: How does metrics analysis contribute to a more comprehensive analysis of code smells in quantum programming? . . . 203

6.5.4 RQ3: Does a model trained on a dataset with both structural and semantic features perform better than one trained on only structural or semantic metrics? . . . 205
6.5.5 RQ4: How do various ML classifiers contribute to the classification of code smells? . . . 207
6.5.6 RQ5: How does statistical analysis validate the dataset's efficacy for accurately detecting code smells? . . . 212
6.5.7 RQ6: How does the proposed methodology compare to existing state -of-the-art approaches in detecting quantum code smells? . . . 213
6.6 Discussion . . . 214

7 QNet: Exploring Deep Learning for Quantum Code Smell Detection 215
7.1 Introduction . . . 215
7.2 Experimental Framework . . . 218
7.2.1 Overview . . . 218
7.2.2 Dataset Preparation . . . 218
7.2.3 Dataset Pre-processing . . . 221
7.2.4 Analysis of DL Models . . . 222
7.3 Results Specific to RQs . . . 224
7.3.1 RQ1: How do QCS influence the frequency of errors in Quantum computers? . . . 224
7.3.2 RQ2: What is the relative impact of different QCS on the performance of QC systems? . . . 225
7.3.3 RQ3: Do the outcomes of DL models, developed with distinct hyperparameters, exhibit complementary behavior? . . . 226
7.3.4 RQ4: What are the predictive capabilities of different DL models developed for QCS detection? . . . 229
7.3.5 RQ5: What is the effectiveness of the proposed model when compared with well-known DL models and pre-trained transfer learning model? . . . 230
7.4 Discussion . . . 232

8 m-QIEA: A Quantum-Inspired Detection Approach 233
8.1 Introduction . . . 234
8.2 Experimental Framework . . . 235
8.2.1 Dataset . . . 236
8.2.2 Phase 1: Feature Extraction . . . 238

8.2.3 Phase 2: Best Feature Selection . . . 239
8.2.4 Phase 3: Classification . . . 247
8.3 Results . . . 248
8.3.1 Setup . . . 248
8.3.2 Research Questions . . . 249
8.3.3 Results Specific to RQs . . . 249
8.3.4 Ablation and Sensitivity Analysis . . . 261
8.3.5 Interpretability of the Selected Feature Subsets . . . 264
8.4 Discussion . . . 265

9 A Comparative Analysis of Quantum Programming Bug Detection Tools 267
9.1 Introduction . . . 267
9.2 Tools Used . . . 268
9.2.1 QChecker . . . 270
9.2.2 QSmell . . . 270
9.2.3 QPAC . . . 270
9.3 Proposed Methodology . . . 271
9.3.1 Data Pre-processing . . . 271
9.3.2 Evaluation Measures Selected . . . 272
9.4 Results . . . 273
9.4.1 Tool Bug Detection Assessment . . . 273
9.4.2 Tool Reliability Assessment . . . 275
9.4.3 Performance Metrics . . . 275
9.4.4 Tool Usability Assessment . . . 275
9.5 Discussion . . . 276

10 Conclusion, Future Work and Social Impact 277
10.1 Summary of Work . . . 277
10.2 Applications of the Proposed Work . . . 280
10.2.1 Industrial Applications . . . 280
10.2.2 Research Applications . . . 281
10.3 Limitations of the Work . . . 282
10.4 Future Work . . . 283

Appendix 285

List of Publications and their Proofs 291

Plagiarism Certificate and Report **297**

Supervisors' Biography **299**

Author's Biography **301**

References **302**

List of Tables

1.1 Difference between Design Antipattern and Code Smell 11

2.1 Classification of software antipatterns along with their identification criteria and severity indicators. 43

3.1 Code Smells Identified by Wake in 2003 93
3.2 Research Questions Imposed on this Systematic Literature Review 95
3.3 Layer-wise Article Selection Process 99
3.4 Questions for Quality Assessment 102
3.5 Data Extraction Form 103
3.6 List of Code Smells Detected in the Papers Analyzed under this SLR 105
3.7 Logical Representation of Code Smells using Metrics 109
3.8 Evaluation metrics reported in primary studies 119

4.1 Dataset Description with Training and Testing Splits and Smell Category . . 130
4.2 Performance comparison of base classifiers 139
4.3 Sensitivity Analysis of DIR-SMOTE Hyperparameters on jEdit Dataset . . . 144
4.4 Performance Metrics for Different Sampling Techniques for Apache 145
4.5 Performance Metrics for Different Sampling Techniques for jEdit 146
4.6 Performance Metrics for Different Sampling Techniques for DesigniteJava . 147
4.7 Performance Metrics for Different Sampling Techniques for EDTforCSD . . 148
4.8 Performance Metrics for Different Sampling Techniques for MLCQ 149
4.9 Traceable linkage between SHAP importance and the perturbation δ_i across two scenarios. 150
4.10 Model-agnostic evaluation of DIR-SMOTE on augmented datasets 152
4.11 AutoML-Driven Classifier Selection after DIR-SMOTE Resampling 153
4.12 Ablation Study: F1-Score Comparison of DIR-SMOTE Variants on jEdit Dataset 154
4.13 Wilcoxon Signed-Rank Test Results for DIR-SMOTE vs. Existing Techniques 155
4.14 Effect Size (Cohen's d) of DIR-SMOTE over Competing Resampling Methods 155
4.15 End-to-end synthetic sample generation from instance `Apache_C1023`. . . 158

5.1 Impact of Code Metrics on Diverse Code Smells 164
5.2 Performance Metrics of Homogenous Ensembles 166

5.3 Performance Metrics of Heterogenous Ensembles . . . 166
5.4 Statistical Pairwise Comparison among ML Classifiers . . . 168
5.5 Pairwise Statistical Comparison for Every Code Smell . . . 169
5.6 Dataset after source resolution. Instances are class-release pairs, groups are distinct class names. . . . 173
5.7 Re-run baselines and the hyperparameter ranges searched . . . 174
5.8 Within-project results, macro-averaged over five smells (mean of ten repetitions) . . . 175
5.9 Per-smell F-measure and MCC for the three strongest approaches . . . 175
5.10 Wilcoxon tests of SmellGraph against each baseline family on MCC (Holm-corrected) . . . 176
5.11 Ablation study (within-project, macro values) . . . 176
5.12 Leave-one-project-out results over 30 projects (macro values) . . . 177

6.1 Overview of Common Quantum Gates (continued) . . . 182
6.2 Specific Quantum Functions and Formulas for Calculating Quantum Metrics 185
6.3 Description of Instances for Each Code Smell in Quantum Dataset . . . 196
6.4 Base ML Classifiers Selected in this Study . . . 196
6.5 Calculation of Quantum Metrics for Bernstein-Vazirani Circuit . . . 199
6.6 Metric Conditions and Computed Values for Code Smell Detection in Bernstein Vazirani Circuit . . . 200
6.7 Code Smells Validation Questionnaire . . . 201
6.8 Individual Reviewer Ratings for Validity and Effectiveness of Code Smells . 202
6.9 Survey results per smell . . . 203
6.10 Descriptive Statistics for Quantum Metrics . . . 203
6.11 Correlation Analysis Results of Metrics . . . 205
6.12 Performance of Different Feature Sets for Code Smell Detection . . . 205
6.13 Performance of Various Machine Learning Models on Different Code Smells 206
6.14 Classifiers statistical performance results Across Different Smell Types . . . 211
6.15 Classifiers paired comparisons results . . . 211
6.16 Comparison of state-of-the-art Approaches on Common Code Smells . . . 213

7.1 Evaluation of Individual models . . . 226
7.2 Evaluation of Combination of Models . . . 227
7.3 Architectural Details of the Proposed Model . . . 229

8.1 Summary of Datasets Used in the Study . . . 236

8.2 Configuration of the Selected Models . . . 239
8.3 Comprehensive Feature Extraction Performance Across Classical Datasets . 240
8.4 Comprehensive Feature Extraction Performance Across Quantum Datasets . 241
8.5 Direction rule for the m-QIEA rotation update. The condition $F(X_i) < F(B)$ indicates that the stored best solution has better fitness than the current observed solution. . . . 245
8.6 Performance Comparison of Machine Learning Classifiers Across All Benchmark Datasets . . . 247
8.7 Research Questions and Their Descriptions . . . 249
8.8 Performance comparison of metaheuristic algorithms on classical datasets. . 250
8.9 Performance comparison of metaheuristic algorithms on quantum datasets. 250
8.10 Effect of Feature Count on Specific Code Smell Detection . . . 252
8.11 Dataset-level paired t-test results comparing m-QIEA with the baseline feature-selection algorithms using SVM-wrapper accuracy. . . . 254
8.12 t-test Results: m-QIEA vs. Other Feature Selection Techniques . . . 254
8.13 Statistical Pairwise Comparison among Feature Selection Techniques . . . 254
8.14 Comparative Analysis of Feature Selection vs. LLM-Based Code Smell Detection on Quantum and Classical Datasets . . . 256
8.15 Performance of the proposed approach on the large-scale SmellyCode++ dataset. . . . 258
8.16 Multi-language evaluation of m-QIEA using average F1-score. . . . 259
8.17 Performance of m-QIEA on the original and large-scale quantum datasets. . 260
8.18 Sensitivity of m-QIEA to the maximum number of optimizer iterations. . . . 261
8.19 Sensitivity of m-QIEA to the dimensionality of the learned feature representation. . . . 262
8.20 Component-wise ablation of the proposed m-QIEA. . . . 263
8.21 Compactness of representative m-QIEA-selected feature subsets. . . . 264

9.1 Summary of Diverse Bugs Identified by Detection Tools . . . 269
9.2 Comparison of Quantum Bug Detection Tools on Bugs4Q Dataset . . . 274
9.3 Performance Metrics for Each Tool . . . 275
9.4 Performance of Detecting tools on various quantum languages . . . 276

A.1 List of Primary Studies Selected for SLR . . . 285
A.2 Detailed Description of Selected Studies for the Review . . . 287
B.1 Project-Wise Code Smell values from QNet . . . 289

List of Figures

1.1 Types of Software Design Patterns 8
1.2 Applications of Antipattern Detection in the Real World 19

2.1 Research Process 33
2.2 ROC- Test Amplitude Estimator circuit 48
2.3 LC- Load Qasm circuit 49
2.4 IM- Stochastic Swap circuit 50
2.5 IdQ- Parametrized circuit 51
2.6 IQ- Bernstein Vazirani circuit 52
2.7 Flow of Experimental Design 79
2.8 Illustration of Ten-Fold Cross Validation 86

3.1 Contribution of Top 9 Active Countries in Code Smell Research 92
3.2 Code Smell Categorization given by Mantyla 92
3.3 Kitchenham Strategy for SLR 94
3.4 Related Terms for the Review 96
3.5 Article Selection Process 98
3.6 Year-wise Publication in Last Ten Years 99
3.7 Article Published in various categories 101
3.8 Available Tools with their Detection Mechanism 109
3.9 Commonly used Deep Learning Algorithms 113
3.10 Commonly used Datasets in Detecting Code Smells 116
3.11 Evaluation Metrics Used for Result in Primary Studies 118
3.12 Performance analysis for Independent Variables 120
3.13 Performance metrics for different DL models 121
3.14 Performance metrics for different Training Strategies 122

4.1 Proposed Methodology for Enhancing Code Smell Detection Using Sampling Techniques and Different Learning Paradigms 129
4.2 SHAP Summary Plot for DIR-SMOTE Technique on EDTforCSD Dataset . . 139
4.3 SHAP Summary Plot for DIR-SMOTE Technique on MLCQ Dataset 140
4.4 SHAP Summary Plot for DIR-SMOTE Technique on Apache Dataset 140
4.5 SHAP Summary Plot for DIR-SMOTE Technique on jEdit Dataset 140
4.6 SHAP Summary Plot for DIR-SMOTE Technique on DesigniteJava Dataset . 141

4.7 Density estimation techniques for weighting minority samples across five datasets. . . . 143

5.1 Overview of Proposed Methodology . . . 162
5.2 Training Accuracy Comparison of ML Classifiers . . . 167
5.3 Performance Comparison of Different ML Classifiers . . . 167
5.4 ROC Curve Comparison of ML Classifiers . . . 168
5.5 Flow of Method for SmellGraph . . . 170

6.1 An Overview of the Proposed Methodology for QUAMET . . . 184
6.2 An Example depicting Long Circuit Code Smell . . . 189
6.3 An Example depicting Excessive Measurement Code Smell . . . 190
6.4 An Example depicting Unconnected Circuit code smell . . . 191
6.5 An Example depicting Inconsistent Bits code smell . . . 192
6.6 An Example depicting Low Quantum Volume code smell . . . 194
6.7 Insufficient Depth per Qubit . . . 195
6.8 Example Bernstein Vazirani Quantum Circuit to show metric values . . . 199
6.9 Outliers in Quantum Metrics . . . 204
6.10 Box plots of the performance (accuracy) exhibited by the considered machine learning models for all the six smells. . . . 209
6.11 ROC AUC of Classifiers from best to worst for different code smells detection . . . 210

7.1 Overview of the Proposed Approach . . . 219
7.2 Converting Quantum Code to Execution Matrix . . . 220
7.3 Values of QCS metrics for sample projects . . . 221
7.4 Analysis of Source Code Lengths . . . 223
7.5 Error Produced due to Decoherence of Qubits . . . 225
7.6 ROC curve comparison among DL models . . . 230
7.7 A Comparative Analysis of Results Between the Proposed Model and Other Models . . . 231

8.1 Overview of Proposed Methodology for Code Smell Detection . . . 235
8.2 Different mutation rates on fitness score and convergence speed . . . 242

9.1 Overview of the Proposed Methodology . . . 271
9.2 Comparison of True and False Positives/Negatives in Detection Tools . . . 275

PUBLICATIONS

Papers Published in International Journals

1. Ruchika Malhotra, Bhawna Jain, and Marouane Kessentini. "Examining deep learning's capability to spot code smells: a systematic literature review." **Cluster Computing** 26.6 (2023): 3473-3501. https://doi.org/10.1007/s10586-023-04144-1.

2. Ruchika Malhotra, Bhawna Jain, and Marouane Kessentini. "QNet: exploring deep learning for quantum code smell detection." **Software Quality Journal** 33.2 (2025): 25. https://doi.org/10.1007/s11219-025-09724-5

3. Ruchika Malhotra, Bhawna Jain, and Marouane Kessentini. "DIR-SMOTE: A Density-Influence Resampling Strategy for Code Smell Detection in Imbalanced Datasets". **Automated Software Engineering** 33,76 (2026) https://doi.org/10.1007/s10515-026-00624-x.

4. Ruchika Malhotra, Bhawna Jain, and Marouane Kessentini. "m-QIEA: A Quantum-Inspired Evolutionary Algorithm for Optimized Feature Selection in Code Smell Detection". **Journal of Systems and Software** (2026) : 113115 https://doi.org/10.1016/j.jss.2026.113115.

Papers Communicated in International Journals

1. Ruchika Malhotra, Bhawna Jain, and Marouane Kessentini. "QUAMET: Structural and Semantic Metrics for Quantum Code Smell Detection". **Quantum Information Processing, Springer**.

Papers Published in International Conferences

1. Ruchika Malhotra, Bhawna Jain, and Marouane Kessentini. "An Empirical Study on Feature Extraction Using Ensemble Deep Learning for Code Smell Detection," *International Conference on Business Intelligence and Data Analytics (BIDA 2024)*. **(Scopus, Accepted and Presented, 06 April 2024)**

2. Ruchika Malhotra, Bhawna Jain, and Marouane Kessentini. "A Comparative Analysis of Quantum Programming Bug Detection Tools," *5th International Conference on Data Science and Applications (ICDSA 2024)*. **(Scopus, Accepted and Presented, 17 July 2024)**

Paper Communicated in International Conference

1. Ruchika Malhotra, Bhawna Jain, and Marouane Kessentini."SmellGraph: A Hybrid Graph Neural Network Framework for Multi-View Code Smell Detection,"*7th International Conference On Computational Intelligence (ICCI 2026)*. **(Scopus, Communicated)**

List of Abbreviations

Acc Accuracy

ADASYN Adaptive Synthetic Sampling

AI Artificial Intelligence

AMW Average Method Weight

ANN Artificial Neural Network

API Application Programming Interface

AST Abstract Syntax Tree

ATFD Access to Foreign Data

AUC Area under Receiver Operating Characteristic (ROC) Curve

AutoML Automated Machine Learning

BERT Bidirectional Encoder Representations from Transformers

Bi-LSTM Bidirectional Long Short-Term Memory Network

BNB Bernoulli Naive Bayes

BUR Base Class Usage Ratio

CART Classification And Regression Trees

CBO Coupling Between Objects

CDEN Comment Density

CDISP Coupling Dispersion

CDUP Code Duplication

CHURN Code Churn

CI/CD Continuous Integration / Continuous Deployment

CINT Coupling Intensity

CFG Control Flow Graph

CM Change Metric Spread

CNN Convolutional Neural Network

CPG Code Property Graphs

CYCLO Cyclomatic Complexity

DIR-SMOTE Density and Influence-based Resampling using Synthetic Minority Oversampling Technique (SMOTE)

DL Deep Learning

DT Decision Tree

DIT Depth of Inheritance Tree

EGAPSO Euclidean distance-based Genetic Algorithm and Particle Swarm Optimization

ENN Edited Nearest Neighbours

FANIN Fan-In Metric

FANOUT Fan-Out Metric

FDP Foreign Data Providers

FS Feature Selection

GA Genetic Algorithm

GB Gradient Boosting

GAT Graph Attention Network

GCN Graph Convolutional Network

GNB Gaussian Naive Bayes

GNN Graph Neural Network

GRU Gated Recurrent Unit

IoT Internet of Things

IDE Integrated Development Environment

KDE Gaussian Kernel Density Estimation

KNN K-Nearest Neighbour

LAA Locality of Attribute Accesses

LB Lack of Balance

LCOM Lack of Cohesion of Methods

LDA Linear Discriminant Analysis

LIME Local Interpretable Model-Agnostic Explanations

LLM Large Language Model

LOC Lines of Code

LOF Local Outlier Factor

LR Logistic Regression

LSTM Long Short Term Memory

LSA Latent Semantic Analysis

MAXNESTING Maximum Nesting Level

MCC Matthews Correlation Coefficient

ML Machine Learning

MLP MultiLayer Perceptron

MNB Multinomial Naive Bayes

m-QIEA modified Quantum-Inspired Evolutionary Algorithm

MSE Mean Squared Error

NOC Number of Children

NOAM Number of Added Methods

NOAV Number of Accessed Variables

NOM Number of Methods per Class

NOP Number of Parents

OO Object Oriented

P Precision

PAR Number of Parameters

PCA Principal Component Analysis

PDG Program Dependence Graph

PSO Particle Swarm Optimization

QC Quantum Computing

QCS Quantum Code Smell

QIEA Quantum Inspired Evolutionary Algorithm

QML Quantum Machine Learning

QP Quantum Program

R Recall

RF Random Forest

RBF Radial Basis Function

RFC Response for a Class

RMSE Root Mean Square Error

RNN Recurrent Neural Network

ROC Receiver Operating Characteristic

RQ Research Question

RUS Random Undersampling

SEC Static Elements Count

SDC Static Data Complexity

SGD Stochastic Gradient Descent

SHAP SHapley Additive exPlanations

SLR Systematic Literature Review

SMOTE Synthetic Minority Oversampling Technique

SVM Support Vector Machines

TCC Tight Class Cohesion

TL Tomek Links

TPOT Tree-based Pipeline Optimization Tool

WMC Weighted Methods per Class

XAI Explainable Artificial Intelligence

XGB Extreme Gradient Boosting

Chapter 1

Introduction

Software systems have become the backbone of modern society, driving innovation across industries from healthcare to Quantum Computing (QC). As these systems grow in scale and complexity, the increasing complexity of modern software systems presents significant challenges for developers in ensuring scalability, maintainability, and reduced development time [1]. As these systems evolve to meet growing user demands and rapid technological advancements, maintaining software quality becomes vital for sustainable development [2]. While design patterns provide proven, reusable solutions to common architectural problems, the presence of antipatterns introduces risks that compromise software maintainability and quality [3].

Antipatterns represent the "dark patterns" of software design, seemingly convenient shortcuts that accumulate technical debt over time. Antipatterns, or recurring "bad practices," may initially seem effective but ultimately result in increased technical debt, reduced adaptability, and higher maintenance costs [4, 5]. They often manifest as poorly structured code, flawed architectural decisions, or inefficient processes, leading to challenges such as greater debugging effort, higher error risk, and difficulty in extending functionality [6]. Addressing antipatterns early in the development lifecycle is critical for building maintainable and adaptable systems [7, 8].

This thesis addresses the critical research question: Can ML techniques reliably predict antipattern emergence to enhance software maintenance? The thesis explores the impact of antipatterns on software maintainability and proposes strategies to mitigate their effects. By identifying antipatterns early and leveraging design patterns as corrective measures, the work aims to reduce technical debt and enhance software adaptability in both classical and quantum systems. Additionally, the thesis emphasizes the role of automated tools and predictive models in detecting antipatterns, enabling proactive improvements in software quality [9, 10, 11, 12, 13].

The remainder of this chapter presents the foundational concepts and motivations

for this research. Section 1.1 introduces software maintainability and emphasizes its significance. Section 1.2 discusses the steps involved in Empirical Study research and role of predictive modeling in improving software maintainability. Section 1.3 provides an overview of design patterns, while Section 1.4 describes Antipatterns, including their classifications and impact on maintainability. Section 1.5 reviews existing work on antipattern detection, covering both classical and quantum-based approaches. The objective of this research is outlined in Section 1.6 describing motivation, gaps, goals and the research objectives. Finally, Section 1.7 highlights the major contributions and overview of this thesis, and Section 1.8 provides an overview of its organization.

1.1 Software Maintenance

Software maintenance involves the continuous effort to ensure that a software system remains functional, efficient, and adaptable post-deployment. It constitutes 60-80% of total software costs, making it the most expensive phase of the software lifecycle. Unlike initial development, maintenance extends the system's lifecycle by addressing bugs, enhancing performance, and incorporating new features. Effective maintenance significantly influences operational efficiency and the overall user experience. Design patterns simplify maintenance by offering structured, reusable solutions, while antipatterns can escalate maintenance costs and complexities.

1.1.1 Types of Software Maintenance

Software maintenance is categorized into four primary types, each addressing specific challenges:

1.1.1.1 Adaptive Maintenance

Adaptive maintenance becomes necessary when there are changes in the external environment in which the software operates. These changes might include updates to the hardware, operating system, supporting software (such as compilers or libraries), or integration with other systems. As the technological environment evolves, the software must be adapted to remain compatible and functional. Adaptive maintenance ensures that the software continues to operate correctly and efficiently in new or altered environments, thereby extending its useful life.

1.1.1.2 Corrective Maintenance

Corrective maintenance involves identifying and fixing errors or defects in the software that become apparent during its use. These issues may stem from flaws introduced during the development phase or from unforeseen usage scenarios. Such maintenance is often urgent, as the errors can disrupt the system's operation or lead to incorrect results. Corrective actions are aimed at restoring the software to its intended functionality and typically include debugging, error correction, and quick fixes. This type of maintenance is critical for maintaining system reliability and user trust.

1.1.1.3 Perfective Maintenance

Perfective maintenance focuses on improving or enhancing a software system to meet evolving user requirements or to improve system performance and maintainability. This type of maintenance does not address defects but instead involves updating, modifying, adding, or removing functionalities based on user feedback or business needs. It may also include restructuring the code to make it more understandable, optimizing performance to make the software run faster or use resources more efficiently, and improving documentation. The main goal is to enhance the usability and effectiveness of the software over time, keeping it aligned with current expectations.

1.1.1.4 Preventive Maintenance

Preventive maintenance is carried out to detect and correct potential problems before they become actual faults. This proactive approach includes activities such as code refactoring, removing obsolete or redundant code, updating outdated libraries or dependencies, and improving the system's security and performance. The objective is to enhance the long-term stability, efficiency, and maintainability of the software. By investing in preventive maintenance, organizations can reduce the frequency and cost of future corrective actions and ensure smoother operation over time.

1.1.2 Software Maintainability and its Importance

Software maintainability refers to the ease with which a software system can be modified to correct faults, improve performance, or adapt to changing environments. According to IEEE, it is defined as "the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment."

Maintainability plays a crucial role in determining the long-term sustainability and cost-effectiveness of software systems. A highly maintainable codebase facilitates easier debugging, faster updates, and smoother adaptation to user requirements. As systems grow in size and complexity, ensuring maintainability becomes increasingly vital to avoid technical debt and performance degradation.

In the context of agile and iterative development environments, maintainability ensures rapid delivery of features without compromising system integrity. Antipatterns and poor design decisions, if left unchecked, can significantly degrade this maintainability over time.

1.2 Empirical Study

Empirical study refers to the systematic process of collecting, analyzing, and interpreting real-world evidence to gain insights or validate hypotheses. In software engineering, empirical studies are essential to understanding practices, evaluating tools or models, and improving processes based on observed data rather than assumptions. These studies offer a robust foundation for evidence-based decision-making and contribute to the development of reliable and replicable knowledge.

One of the widely used analytical techniques within empirical studies is **predictive modeling**. Predictive modeling involves using historical or observational data to build statistical or ML models capable of forecasting future trends or classifying new data. It serves as a powerful tool in software engineering to anticipate risks, estimate efforts, detect defects, or evaluate quality attributes. Rather than being a separate methodology, predictive modeling is often integrated within the broader empirical research process as part of the analysis and interpretation phases.

1.2.1 Steps in Empirical Study

Empirical studies typically follow a structured and systematic approach, ensuring the reliability and validity of the results. The general steps are outlined below:

- **Formulation of Research Objectives:** Clearly define the goals of the study and formulate specific research questions or hypotheses. This step lays the foundation for all subsequent activities.
- **Study Design:** Select an appropriate empirical method such as a case study, controlled experiment, or survey. Determine the variables, metrics, and data sources to be used. The design should ensure traceability and reproducibility.

- **Data Collection and Preparation:** Empirical studies rely on real-world data, which may be collected from repositories, tools, surveys, logs, or direct observations. Collected data is preprocessed by cleaning, transforming, and selecting relevant features to ensure consistency and relevance.
- **Model Development and Analysis:** If predictive modeling is employed, this step involves selecting suitable algorithms (e.g., regression, classification, ensemble methods) and training models using the prepared dataset. The goal is to uncover patterns or relationships that can generalize to unseen data.
- **Evaluation and Validation:** Assess the performance of models or hypotheses using appropriate validation techniques such as cross-validation, hold-out testing, or statistical significance testing. Performance metrics (e.g., accuracy, precision, recall, F1-score, etc.) are used to determine the model's effectiveness.
- **Interpretation of Results:** The outcomes are analyzed to address the research questions and understand the practical implications. If predictive models are used, interpretability tools (e.g., SHapley Additive exPlanations (SHAP), Local Interpretable Model-Agnostic Explanations (LIME)) may assist in explaining the contribution of input variables.
- **Conclusion and Reporting:** Summarize the findings, highlight their significance, acknowledge limitations, and propose directions for future work. Proper documentation and reporting ensure the transparency and reproducibility of the study.

This empirical process, when rigorously followed, allows researchers to derive conclusions that are both credible and generalizable. Predictive modeling, when applied thoughtfully, enhances the explanatory and forecasting power of empirical investigations.

1.2.2 Predictive Modeling for Software Maintainability

Software maintainability refers to the ease with which a software system can be modified to correct defects, improve performance, or adapt to changing requirements. As software systems grow in complexity, ensuring high maintainability becomes essential for reducing long-term costs and improving system longevity. Predictive modeling has emerged as a powerful approach for proactively assessing maintainability using historical and structural data from software artifacts.

In this context, predictive modeling involves building ML or statistical models that estimate maintainability metrics or predict maintainability-related issues such as code smells, technical debt, or defect-prone components. These models learn from existing

datasets that capture software properties, such as code complexity, cohesion, coupling, documentation quality, and historical change data.

Predictive modeling not only supports early detection of maintainability concerns but also enables data-driven decision-making in software quality assurance. By integrating predictive modeling within empirical studies, researchers and practitioners can quantify maintainability more effectively and develop tools and recommendations that improve software design and maintenance practices.

1.2.3 Issues in Predictive Modelling

- **Data Quality and Availability:** Models often suffer due to noisy, incomplete, or imbalanced datasets. Limited data volume and issues like data leakage further compromise reliability. Code smell datasets are often imbalanced, with far fewer smelly instances compared to clean code. Additionally, inconsistent labeling and lack of standardized benchmarks introduce noise and hinder model training. In open-source projects, variations in coding style also impact feature reliability.

- **Model Selection and Complexity:** Choosing an appropriate model involves balancing accuracy and interpretability. Overfitting, underfitting, and opaque model behavior are common concerns. DL models can capture complex code semantics but are often black boxes. Traditional models offer interpretability but may fail to generalize across projects. Selecting the right balance between complexity and explainability remains a key concern for real-world adoption.

- **Feature Engineering:** Effective feature selection and transformation are critical. Irrelevant or redundant features can degrade performance and increase model complexity. Crafting meaningful features from source code requires understanding static and semantic structures, such as method calls, inheritance, or cohesion metrics. Redundant or manually extracted features may dilute model performance, especially when dealing with large-scale repositories.

- **Evaluation and Validation:** Using improper metrics or neglecting robust validation techniques, such as cross-validation, can lead to misleading performance assessments. Metrics like accuracy may be misleading due to class imbalance. Precision, recall, F1-score, and Area under ROC Curve (AUC) are better suited for assessing performance. Project-wise or release-wise cross-validation is essential to ensure that models generalize across diverse codebases.

- **Domain Knowledge Gaps:** Lack of subject matter expertise may result in suboptimal modeling choices. Moreover, rapidly evolving domains demand continuous

model adaptation. Understanding code smells requires familiarity with software design principles. Without sufficient domain expertise, automated approaches may flag false positives or miss subtle design flaws. Moreover, evolving development practices demand that models stay updated with new types of smells and antipatterns.

1.3 Design Patterns

Programmers often feel that codes should only be written in a way that a computer can comprehend them, however, it's also a good idea to construct codes that are also clear to humans. Making quality code facilitates keeping these codes up for a longer time. Using design patterns is an efficient option when designing code according to best practices. Gamma et al. [14], in 1994, first gave a brief description of design patterns as follows:

Design patterns are recurring solutions to design problems in specific contexts. They record best practices and give answers to typical design problems, giving developers a channel for sharing effective design tactics.

Design patterns provide simple, reusable solutions to common problems in software design and code formation. Various types of software design patterns are described in Figure 1.1. Software engineers may think more abstractly and make wise design decisions thanks to design patterns, which offer a higher level of vocabulary for software design. Developers may construct adaptable, reusable, and maintainable software systems by applying design patterns. Design patterns enhance code readability, flexibility, and maintainability when applied appropriately. However, the successful application of these patterns is not arbitrary, it is rooted in fundamental software design principles. Among these, the SOLID principles stand out as a comprehensive set of guidelines that complement and reinforce the intent behind many design patterns. Understanding these principles is therefore essential before employing patterns effectively.

1.3.1 SOLID Principles

The SOLID principles represent a set of five fundamental object-oriented design guidelines that aim to improve the flexibility, maintainability, and readability of software systems. These principles, proposed by Robert C. Martin in the early 2000s [15], have since become widely adopted best practices in software engineering.

A brief overview of the SOLID principles is as follows:

1. **Single Responsibility Principle (SRP):** A class should be responsible for a single functionality or purpose. Adding unrelated methods or behaviors violates this prin-

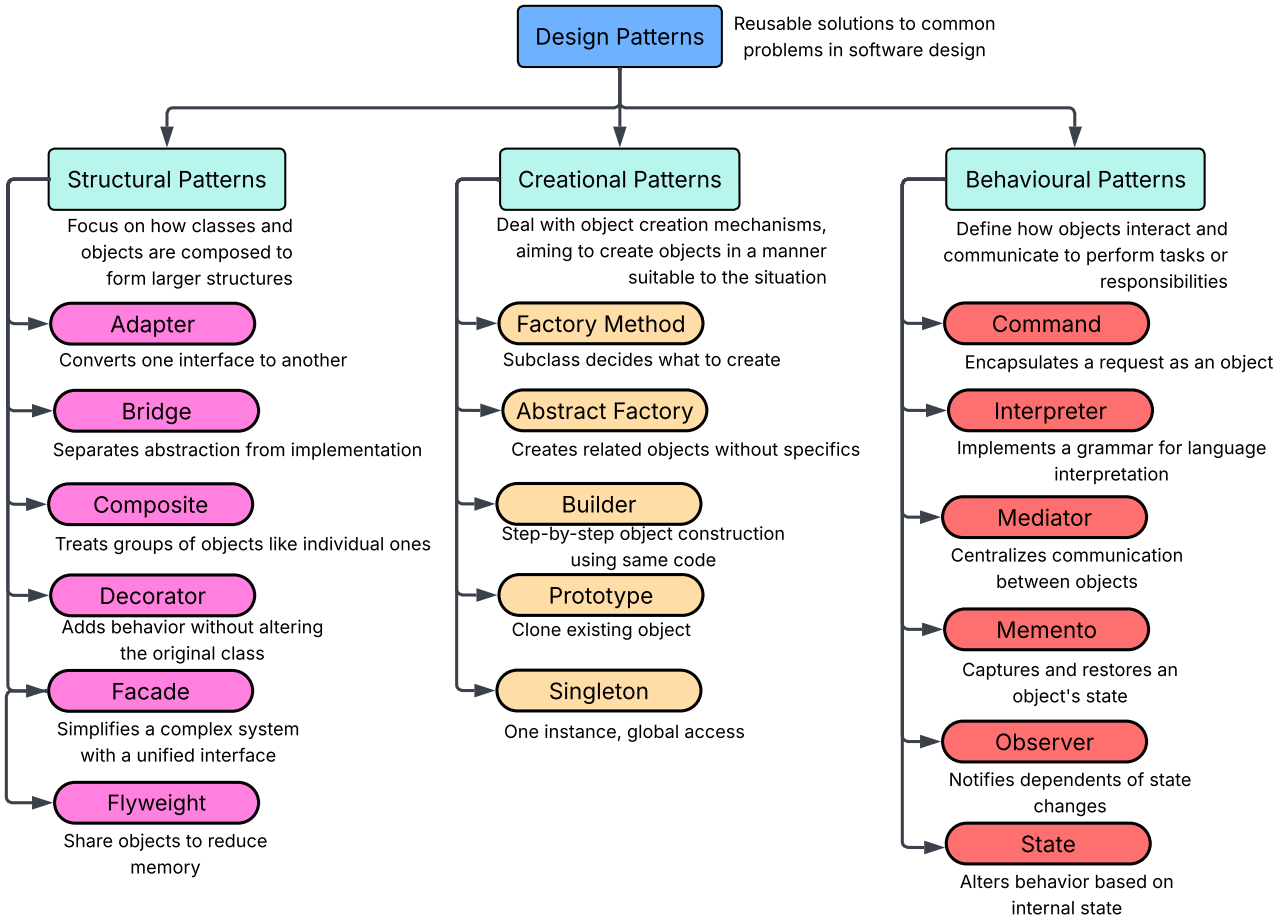


Figure 1.1: Types of Software Design Patterns

ciple. Classes adhering to SRP have only one reason to change, which makes them easier to understand, maintain, and less prone to unintended side effects.

2. **Open–Closed Principle (OCP):** Software entities such as classes, modules, and functions should be open to extension but closed to modification. In practice, this means new features can be added without altering existing source code, thereby reducing the risk of introducing defects.

3. **Liskov Substitution Principle (LSP):** Objects of a subclass should be replaceable with instances of their superclass without affecting program correctness. This principle ensures that derived classes preserve the expected behavior of their base classes, supporting robust and consistent inheritance.

4. **Interface Segregation Principle (ISP):** Clients should not be forced to depend on methods they do not use. Instead of large, generalized interfaces, smaller and more

specific ones should be defined to better align with client requirements and minimize unnecessary dependencies.

5. **Dependency Inversion Principle (DIP):** High-level modules should not depend directly on low-level modules; both should rely on abstractions. This principle promotes decoupling by encouraging the use of interfaces or abstract classes, making systems more modular and adaptable to change.

Adhering to these principles results in software systems that are easier to maintain, extend, and test, while also enhancing modularity and long-term sustainability of the design.

1.3.2 Violation of SOLID Principles

If a code violates the principles of SOLID it can lead to negative consequences in software maintenance. Following are the examples for the same:

1. **Increased complexity:** It makes code harder to understand. Further, if new developers start working on the piece of code it becomes difficult to maintain this code and make modifications without introducing bugs.

2. **Reduced flexibility:** As the code is difficult to understand, it makes it harder to change and hence requires extensive reworking. These types of code are not very good at working.

3. **Decreased reusability:** As the code cannot be improved, it becomes difficult to reuse it further in projects. It limits code sharing and simultaneously increases code duplication.

4. **Testing issues:** The code lacks atomicity, hence it is not fit to test individually.

5. **Maintenance issues:** Such codes are difficult to maintain. If any bug is found in one part of the code base, it can cause unintended consequences to other areas, requiring additional time, effort, and cost to fix it.

6. **Lower code quality:** Lowers team productivity as a whole.

When codes lack SOLID principles, they develop **Antipatterns**.

1.4 Antipatterns

A most coiled term in software engineering is "no silver bullets" implying "no miraculous or simple strategy, tool, or technique" can solve all software engineering difficulties. This is due to the presence of Antipatterns. Programmers have faced the challenge of convoluted code that leads to inefficiencies from the beginning of coding, but it was not until the mid-1990s that antipatterns were found and popularised in the area of software engineering. Koenig and Gabriel [16] discovered that certain recurrent design and execution decisions in software development resulted in undesirable outcomes and inefficiency. Inspired by their discoveries many researchers in this field have led us to a simple definition for antipatterns:

"*Antipatterns are recurring patterns of poor design, implementation, or architectural choices in software development that appear appealing or reasonable at first but ultimately result in negative consequences such as decreased maintainability, reduced performance, increased complexity, or other adverse effects on software quality*." [17]

Antipatterns are flaws seen at two levels:

- **Design antipatterns:** Antipatterns that are found in the design and architecture of programs. They are more complex than code smells and create more negative consequences if not detected. If any program has a design antipattern it is very difficult to detect it as it impacts the whole program.

- **Code smell:** Code smells or bad smells are distinct symptoms or signs in source code that indicate the presence of deeper issues or design faults. They are not code flaws or mistakes but portions of the code base that should be enhanced or refactored to increase readability, maintainability, and overall quality. They serve as early warning indicators of possible problems or opportunities for improvement.

1.4.1 Design Antipatterns

Design Antipatterns, or architectural antipatterns, refer to issues within the high-level organization of software systems. These issues can arise from improper dependencies between modules, poor layering, inadequate modularization, and overly complex or rigid architectures, impacting the system's scalability, performance, and maintainability. For instance, the God Object and Cyclic Dependency architectural smells commonly appear in large codebases, where tightly coupled components restrict flexibility and increase the difficulty of implementing new features.

Table 1.1: Difference between Design Antipattern and Code Smell

Parameters	Code Smells	Design Antipatterns
Definition	Flaws or mistakes found in the source code of the program	Mistakes found in the design and architecture of the program
Scope	Limited to only a function or a class in the code	Has greater scope, impacting multiple components at a time
Impact	Less impact	More negative impact
Consequences	Detecting on time and refactoring can remove it	If not detected, may lead to distortion of the whole program and the need to create a new program
Examples	Large class, feature envy, long method, etc.	Improper use of inheritance

In architectural smells, the entire software's structure and organization are compromised, often due to an initial failure to design for adaptability or extensibility. The Big Ball of Mud is a prominent example, describing architectures lacking clear structure and consistency, resulting in chaotic, unmanageable systems as development scales. Layering Violation is another common architectural smell where components bypass intended architectural layers, leading to code interdependencies that obscure the separation of concerns and degrade modularity.

To mitigate architectural smells, developers often employ metric-based detection techniques. These techniques use metrics like coupling, cohesion, and code complexity to identify problematic areas. Advanced approaches, such as architectural recovery and refactoring, use dependency analysis to understand and correct the architectural layers and hierarchies, often with tools like Arcan [18], ASPredictor [19], and InSet [20] which automatically detects architectural smells based on predefined rules, metrics and ML. Additionally, researchers are exploring quantum metric-based analysis to better handle the complexity in large-scale systems, leveraging quantum principles for quicker and more effective smell detection in high-dimensional software systems.

1.4.2 Code Smells

Code smells, or design smells, are indicators of internal software issues that can hinder software evolution and maintenance, either at the design level or the code level. The term "smell" highlights the subtle nature of these issues; they don't always lead to program failures or crashes but rather make code harder to understand, maintain, or extend, lowering

overall software quality. These smells are considered violations of core software development principles and were popularized in the late 1990s by Beck et al. [21], the creator of extreme programming, who emphasized the importance of design quality. This concept gained further prominence when Fowler included it in the book [22], establishing "Code Smells" as a common term in programming.

Fowler defines code smells as "symptoms" of deeper design issues that can lead to antipatterns if left unaddressed. Examples include the Long Method smell, where methods grow excessively long, making them hard to manage, and Feature Envy, where one class relies heavily on another class's data, indicating a lack of proper separation of concerns. These differ from architectural smells, which manifest at a broader structural level, whereas code smells focus more directly on readability, modifiability, and the potential for errors.

Detection methods for code smells range from traditional metric-based approaches, such as coupling and cohesion metrics, to advanced ML techniques. Tools like PMD and JDeodorant [23] automatically identify code smells by analyzing syntactical structures, dependencies, and control flows. With recent developments in Explainable Artificial Intelligence (XAI), techniques like SHAP allow developers to interpret the factors contributing to detected smells, particularly useful in large codebases where manual detection would be both time-consuming and error-prone.

1.5 Literature review

Code smell detection is pivotal in maintaining software quality, as it serves to identify hidden design deficiencies that could compromise software maintainability and scalability over time. The early identification of code smells plays a crucial role in guiding refactoring efforts, ensuring that software remains robust, adaptable, and easier to evolve. Researchers have made substantial progress in this domain, employing various techniques to enhance the accuracy and efficiency of code smell detection, thus significantly contributing to software quality improvement practices.

This section explored the diverse strategies and methodologies employed for code smell detection, categorizing them into classical and quantum approaches. These approaches not only reflect the evolution of detection techniques, but also highlight the innovative use of both traditional and cutting-edge technologies, each contributing uniquely to advancing the field of code smell detection. By examining these methods, this study aims to uncover research gaps and to draw inspiration for future exploration and refinement of code smell detection methodologies.

1.5.1 Classical Code Smell Detection

Code smells are a critical concept in software engineering, and are often associated with poor code quality, high maintenance costs, and reduced performance. Introduced by Beck in the 1990s [21], the term gained popularity with Fowler's detailed categorization of code smells and the development of refactoring techniques to address them [22]. Fowler's work laid the foundation for research in code quality and design flaws. Mantyla [24] further categorized these smells into seven broad groups, indicating their pervasive impact on software design.

Research on code smells has continued to evolve. Kessentini et al. [25] highlighted the presence of code smells in software design, even though their frequency is lower compared to source code smells. However, the design-level smells still pose risks for long-term software maintenance. Liu et al. [26] expanded the concept of code smells to include testing, error handling, and formatting, emphasizing the need for clean code and best practices.

In the early 2000s, Wake [27] introduced new code smells focused on mathematical computation. These studies, along with others, reflect the increasing complexity of code smells and the need for sophisticated detection and refactoring methods. Secondary studies, particularly Systematic Literature Review (SLR)s, have been essential in synthesizing the growing body of research on code smell detection. Zhang et al. [28] conducted an SLR covering the state-of-the-art from 2000 to 2009, focusing on the most prominent code smells, design methods, and empirical evidence for the adverse effects of code smells. Alsolai et al. [29] expanded the scope by examining DL methods for software maintenance prediction, identifying key gaps in the literature, particularly the limited application of ensemble models for prediction. Fernandes et al. [30] conducted an SLR on code smell detection technologies, comparing the accuracy and recall of various tools. Arshad and Rasool [31] also reviewed antipattern detection tools, offering practical suggestions for researchers working on code smell detection.

The introduction of ensemble models in ML further revolutionized code smell detection. Alazba and Aljamaan [32] employed stacking ensemble techniques, demonstrating a significant improvement in accuracy (up to 98.66%) when detecting six different code smells. Early ML methods, such as metric-based detection, offered faster and more reliable results compared to manual inspections [33, 34]. However, as the complexity of software grew, the need for advanced ML and DL techniques became apparent. Given the structured nature of source code, recent research has focused on DL and Graph Neural Network (GNN)-based models to capture both syntactic and semantic relationships within programs. Mesbah et al. [35] explored the effectiveness of GNNs for code smell detection, showing that graph-based representations outperform traditional ML techniques

in identifying complex patterns. Similarly, Zhang and Kishi [36] applied three distinct GNN architectures for Large Class code smell detection, demonstrating their ability to model hierarchical relationships in software systems. Graph-based techniques have also been applied to specific code smells, such as feature envy. Yu et al. [37] proposed a GNN-based model for detecting and refactoring feature envy, outperforming rule-based methods by leveraging code structure representations. In a follow-up study [38], they introduced metric-based refinements to improve detection efficiency, highlighting the adaptability of GNN-based approaches. Beyond single-code-smell detection, hybrid models have been developed to optimise software refactoring. Prasad and Srivenkatesh [39] integrated GNNs with reinforcement learning and autoencoders for automated code refactoring, achieving improved maintainability and complexity reduction. Zhang and Kishi [40] leveraged Graph Convolutional Network (GCN)s to detect long method smells, reinforcing the utility of graph-based learning for handling large-scale software systems.

Recent studies, such as those by Sandouke et al. [41], explored the effectiveness of various ML models for detecting specific code smells like Large Class and Long Method. These studies highlighted the superiority of the Random Forest and Decision Tree models. Moreover, DL techniques, including transfer learning, have emerged as promising tools in code smell detection [42, 43, 44]. With DL advancements, these models entered the code smell detection domain. Li and Zhang [45] presented a hybrid model employing multi-level code representation. The code underwent parsing into Abstract Syntax Tree (AST), with GCNs offering predictions at both syntactic and semantic levels. A bi-LSTM network with an attention mechanism analyzed code tokens, showcasing superior performance in single and multi-label code smell detection compared to simpler code representations. Zhang et al. [46] introduced DeleSmell, a DL-based code smell detection approach that addressed incomplete feature extraction and sample imbalance. Using a dataset from 24 real-world projects, they employed a refactoring tool to transform good code into smelly code, generating positive samples. The model combined structural (iPlasma) and semantic features (Latent Semantic Analysis (LSA), word2vec) through a Convolutional Neural Network (CNN) branch and a Gated Recurrent Unit (GRU)-attention branch, with final classification by an Support Vector Machines (SVM). Results demonstrated a significant up to 4.41% improvement in accuracy for detecting brain class and brain method code smells compared to existing approaches, confirming the effectiveness of DeleSmell. Ho et al. [43] introduced DeepSmells for code smell detection, targeting deficiencies in prior research. Their approach employed a deep CNN followed by Long Short Term Memory (LSTM) for context encoding, utilizing deep neural networks with a weighted loss function. Empirical studies on benchmark datasets demonstrated DeepSmells' superior performance in F1 score and Matthews Correlation Coefficient (MCC).

A significant challenge in code smell detection is the issue of class imbalance, where non-smelly instances vastly outnumber smelly ones. To address this, researchers have applied various balancing techniques. Khleel and Nasraldeen [47] combined Bidirectional Long Short-Term Memory Network (Bi-LSTM) and GRU models with random oversampling and tranfer learning to improve accuracy. Pecorelli et al. [48] found that SMOTE significantly enhanced classifier performance, while Zhang et al. [49] applied SMOTE in their Metric-Attention Residual Network-based system (MARS) model for Brain Class code smell detection. Similarly, Zhang et al. [46] introduced the DelSmell model, which replaced non-smelly code with smelly instances to improve detection accuracy. Dewangan et al. [50] applied ensemble stacking algorithms to detect code smells in four datasets, achieving high accuracy through feature selection techniques like Chi-square. Likewise, Boutaib et al. [51] introduced ADIODE, a search-based technique using evolutionary algorithms to detect smells in imbalanced datasets.

With the advent of evolutionary algorithms, the convergence of various techniques became easier for optimization in code smell detection research. Boussa et al. [52] introduced a novel method for software bad-smell detection, utilizing competitive coevolutionary search. It employed two populations: one generated detection rules for code-smell coverage, while the other produced "artificial" code-smells. The approach showed promising performance over single population-based methods in statistical analysis across diverse benchmarks. Saranya et al. [53] aimed to enhance code smell detection using the Euclidean distance-based Genetic Algorithm and Particle Swarm Optimization (EGAPSO). Unlike rule-based methods, EGAPSO relied on similarity measures, eliminating manual threshold determination. Tested on open-source projects for various code smells, EGAPSO outperformed other detection approaches [54, 55]. Ghannem et al. [56] proposed a Genetic Algorithm (GA)-based approach to enhance design smell detection. Unlike rule-based methods, it utilized system-to-defect similarity, removing the need for explicit rules. Tested on four open-source systems, the method effectively identified potential design defects. Caram et al. [57] reviewed DL-based approaches for detecting code smells and found that genetic algorithms were the most frequently used technique, with decision trees and random forests offering the best average performance. Similarly, Azeem et al. [58] focused on heuristic methods for code smell detection but highlighted the need for a more structured adoption of DL techniques.

Despite these advancements, manually selecting classifiers for different datasets remains a challenge. Automated Machine Learning (AutoML) emerged as a solution to this problem, automating model selection, hyperparameter tuning, and evaluation [59]. Al-Fraihat et al. [60] demonstrated the utility of AutoML in detecting code smells, specifically in move method refactoring, by reducing coupling and enhancing cohesion. As the

complexity of code smell detection models increased, the demand for model interpretability grew. XAI became an essential tool for providing transparency into model behavior. Huang et al. [61] applied XAI to code smell prioritization, showing that aligning feature selection with developer expectations improved explanation coverage. In another study, Huang et al. [62] used SHAP values to interpret model predictions, demonstrating that Correlation-based Feature Selection significantly reduced Krippendorff's Alpha in code smell prioritization tasks.

Overall, advancements in DL, ML, evolutionary algorithms, AutoML, and XAI have significantly contributed to improving code smell detection, offering new avenues for handling complex software systems. However, challenges like class imbalance, model transparency, and classifier selection remain areas for future research.

1.5.2 Quantum Code Smell Detection

Code smell detection has primarily been explored in classical computing environments; however, research in Quantum Code Smell (QCS) detection is still in its early stages. Huang and Martonosi [63] introduced quantum program assertions for debugging, targeting challenges like limited access to quantum internal states. Their approach used statistical tests on classical observations to evaluate quantum program states and was supported by a tool integrated into a quantum program simulator. They identified bugs through benchmarking programs and proposed strategies for effectively placing assertions in quantum programs. Proq [64], a more recent advancement, introduced projection-based runtime assertions that leverage quantum logic predicates and local projections to debug programs on hardware with limited measurement capacity.

Campos and Souto [65] emphasized the importance of reproducibility and comparability in quantum software engineering research. They developed QBugs, a benchmark designed to assist researchers in evaluating quantum software studies. QBugs provides experimental subjects and infrastructure, which expedite study assessments and improve reproducibility in the rapidly evolving field of quantum programming. Paltenghi and Pradel [66] analyzed 223 real-world bugs from 18 open-source quantum computing platforms, highlighting that nearly 40% of the bugs were quantum-specific. These errors, often found in abstractions linked to quantum programming, presented unexpected results. They identified eleven new quantum-specific bug patterns, which underscore the importance of addressing vulnerabilities in quantum computing systems. This study serves as a guide for tool developers and platform builders. Similarly, Zhou et al. [67] examined 391 bugs from 22 public repositories of nine popular Quantum Machine Learning (QML) frameworks. They found that 28% of the faults were quantum-specific, necessitating specialized approaches to mitigate them. The study also developed a taxonomy for fault

causes and symptoms in QML platforms and identified significant challenges faced by QML developers, offering recommendations to improve quantum code quality. Openja et al. [68] analyzed 118 open-source quantum projects from GitHub across ten categories, identifying common issues such as code convention violations and inadequate error handling. Their study established a statistically significant correlation between faults in quantum software and factors like code design and redundant code, underlining the need for better maintenance practices to improve quantum software quality.

Kher et al. [69] introduced Q-Diff, an automated framework designed to detect issue-fix patterns in IBM Qiskit quantum code. Using AST, they developed a proof-of-concept tool to identify quantum bug-fix patterns, aiming to raise awareness in the quantum software engineering community about enhancing the quality of quantum software.

Chen et al. [70] contributed a critical advancement by introducing the *QSmell* tool, which was designed to detect eight specific categories of QCS. Their work, focused on quantum computers, marked a significant step forward in the detection of quantum-specific code smells, setting the stage for future developments in quantum code quality assurance. De Stefano et al. [71] contributed with an empirical study revealing how transpilation, especially gate set choices—can significantly alter smell metrics like long circuit, indicating that detection tools must account for aftermath of transpilation.

This review of quantum code smell detection research highlights the field's progression from initial debugging tools to more advanced detection methods. However, further work is needed to refine these techniques and develop new tools to address the complexities of quantum code smells more effectively.

1.6 Objective of the Thesis

This section outlines the core direction of the research by presenting the motivation behind the study, its scope, and the key challenges identified through a review of existing literature. It highlights the research gaps, formulates the problem statement, and defines the vision, focus, and specific goals that guided the work. Together, these elements provide a clear foundation for the methodology and contributions presented in the thesis.

1.6.1 Motivation and Scope

With the exponential growth of mission-critical software systems, spanning classical enterprise architectures, Internet of Things (IoT) ecosystems, Artificial Intelligence (AI)-powered platforms, and emerging quantum applications, maintaining software quality has evolved from being a post-development concern to a foundational requirement. Soft-

ware systems today are becoming increasingly complex, distributed, and hybrid in nature, where even minor degradations in code quality can lead to cascading failures, increased technical debt, and unsustainable maintenance costs. Studies report that technical debt consumes nearly 23–42% of developer productivity, underscoring the urgent need for proactive solutions to ensure long-term maintainability.

Among the most persistent challenges to software maintainability are *code smells*, serving as surface level indicators of deeper design or structural issues that degrade code readability, flexibility, and adaptability. While initial detection strategies relied on heuristic or rule-based systems, their inability to generalize across diverse codebases has led to the adoption of ML and DL for automated code smell detection. Despite notable advances, several limitations persist, including challenges related to data imbalance, subjective interpretation of smells, limited explainability of DL models, and the lack of tools capable of handling code at the scale of modern software repositories.

At the same time, the rise of *QC* introduces new dimensions to the code smell problem. Quantum software development is rapidly accelerating, yet quantum-specific smells such as *Entanglement Overcoupling* or *Qubit Decay Ignorance* remain undefined in traditional taxonomies. Additionally, hybrid classical-quantum systems, which are becoming mainstream in real-world applications, pose unique challenges that existing smell detection tools are ill-equipped to address. The classical computational cost of analyzing large-scale software graphs also makes traditional tools insufficient in this new paradigm, where quantum-enhanced methods could offer asymptotic advantages in path-sensitive and interprocedural analysis.

This thesis is motivated by the need to build scalable, cross-paradigm, and intelligent smell detection frameworks that address these challenges through a combination of DL, resampling techniques, ensemble modeling, graph-based analysis, and QML. The scope of this work encompasses both classical and quantum software, to enhance its applicability across various domains. Figure 1.2 illustrates the wide-reaching impact of advanced code smell detection, from industrial applications to safety critical domains like aerospace and healthcare. To support these domains, this thesis explores both fundamental and applied research dimensions, ensuring theoretical soundness and practical usability for developers working across computing paradigms. The aim is to not only propose novel detection algorithms but also formalizing new smell categories, generating representative datasets, and building tools that are practically usable by developers across different computing environments. Key motivation and scope for this thesis is mentioned below:

- The primary motivation behind this research lies in the increasing complexity of modern software systems and the urgent need to ensure maintainability and code quality across both classical and emerging quantum paradigms.

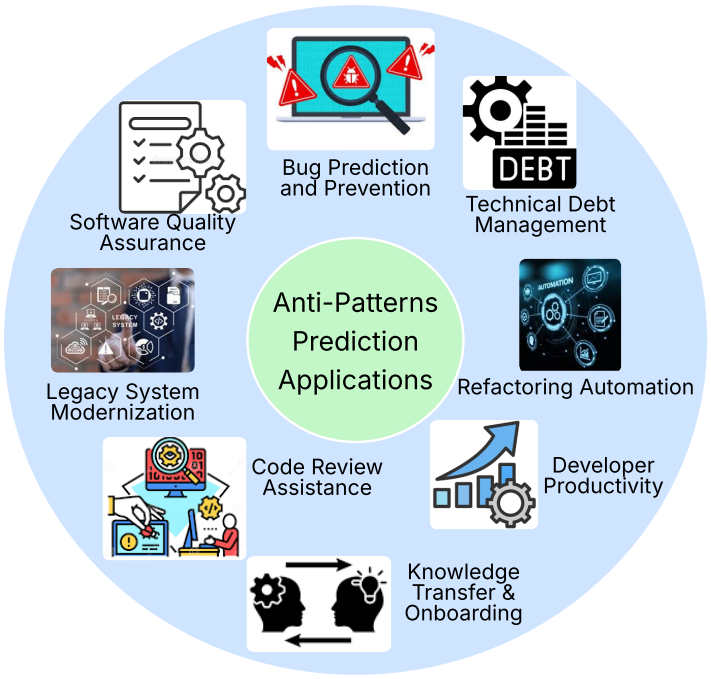


Figure 1.2: Applications of Antipattern Detection in the Real World

- Research in areas such as AI-driven software engineering, static code analysis, quantum programming, and software maintainability contributes to the development of intelligent tools capable of detecting and mitigating design flaws early in the software development lifecycle.

- Code smell detection technologies offer significant impact by enabling early detection of architectural and design issues, reduction of technical debt, improvement in software readability, adaptability, and maintainability, and overall cost-effective software evolution.

- With the increasing adoption of ML and DL in software analytics, there is a need for more accurate, explainable, and automated models that can operate across large, heterogeneous, and imbalanced code datasets.

- On the quantum side, the scope includes proposing new taxonomies of code smells relevant to quantum programming, building and benchmarking datasets for quantum smell detection, and developing hybrid classical–quantum models, including quantum kernel methods and quantum-inspired evolutionary algorithms, to detect complex semantic issues in quantum source code.

- Ensuring software quality in safety-critical domains such as aerospace, healthcare,

and national security necessitates robust and explainable detection frameworks across all programming paradigms.

- The thesis aims to deliver techniques that are generalizable across languages and paradigms, scalable for large repositories, and usable in practical software engineering workflows, contributing to more maintainable and reliable software systems in both classical and quantum domains.

1.6.2 Research Gaps

Despite recent advancements in code smell detection, several open challenges persist across both classical and quantum programming paradigms. A critical review of the literature, along with empirical insights from this research, reveals the following key gaps:

- **Lack of Quantum-Specific Smell Detection Frameworks:** While classical code smells have been extensively studied, quantum programming introduces unique antipatterns such as Entanglement Overcoupling and Qubit Mismanagement. There is no standard taxonomy or detection framework tailored for quantum software, leaving a significant gap in quality assurance for emerging quantum systems.

- **Cross-Paradigm Detection is Underexplored:** With the rise of hybrid systems that integrate classical and quantum codebases, traditional smell detectors fall short. Existing tools are limited to a single paradigm and lack the capability to correlate design issues across hybrid architectures.

- **Imbalanced Datasets Affect Model Generalization:** Most code smell datasets suffer from class imbalance, where faulty classes are underrepresented. This impairs the training of predictive models, leading to biased outcomes. Existing resampling techniques often fail to preserve the semantic integrity of the code.

- **Insufficient Labeled Data for Quantum Smell Detection:** The absence of publicly available, labeled datasets for quantum programs with code smells is a major bottleneck. Without such datasets, supervised learning techniques cannot be effectively applied to quantum smell detection.

- **Limited Interpretability in DL Models:** Although DL methods show strong performance, their black-box nature limits their adoption in practice. Developers need explainable outputs to trust and act upon model predictions, an aspect largely missing from current tools.

- **Scalability Challenges in Large Codebases:** Most proposed techniques are evaluated on small-scale datasets or academic systems. Their ability to detect smells efficiently in real-world, large-scale codebases, spanning millions of lines of code, remains unverified.

Addressing these research gaps is essential for developing scalable, interpretable, and paradigm-agnostic code smell detection systems that are suitable for both classical and quantum software environments.

1.6.3 Problem Statement

This research addresses the pressing challenges in software maintenance caused by the presence of antipatterns, which degrade software quality, increase technical debt, and complicate future enhancements. Existing antipattern detection approaches often suffer from limited scalability, poor generalization across systems, and inadequate handling of minority antipattern classes. Furthermore, traditional detection methods frequently rely on static heuristics or manual inspection, which are inefficient and error-prone in large, complex codebases. The incorporation of machine learning-based techniques introduces new opportunities, yet these approaches face issues related to class imbalance, ineffective feature selection, and lack of interpretability, especially in emerging paradigms like quantum software systems. Hence, the research's problem statement is:

"To design and develop effective, interpretable, and scalable machine learning methodologies for antipattern detection that address data imbalance, improve prediction accuracy, and enhance software maintainability in both classical and quantum programming environments."

1.6.4 Vision

To enhance software maintainability by developing efficient models for antipattern and code smell detection.

1.6.5 Focus

The focus of the thesis is centered on multiple key research objectives, each addressing a critical challenge in the domain of code smell detection and software quality assurance. These objectives have been designed to systematically explore, analyze, and improve the detection of design inefficiencies in both classical and quantum software systems. The research particularly emphasizes the need to address challenges such as class imbalance, feature selection, semantic representation of code, model interpretability, and quantum

software anomalies. The thesis also aims to contribute novel insights into the evolution of software engineering practices by proposing data-driven, scalable, and explainable frameworks that align with modern software development trends. Specifically, the thesis focuses on the following objectives:

- **Objective 1**: To conduct a systematic literature review in order to acquire insight into the tools and techniques utilised in various stages of software code maintenance and evolution.
- **Objective 2**: To investigate and utilize DL algorithms for the purpose of detecting software antipatterns.
- **Objective 3**: To explore and exploit the use of evolutionary algorithms in order to optimize the antipattern detection process.
- **Objective 4**: To evaluate the performance and determine the effectiveness of benchmark detection and correction tools used for software antipatterns.

1.6.6 Goals

To effectively accomplish the intended outcomes of the research, each objective is broken down into targeted goals that define clear actions and milestones. These goals help in systematically navigating the research path, ensuring a focused and comprehensive approach to addressing each objective.

1. **Conduct a systematic literature review in order to acquire insight into the tools and techniques utilised in various stages of software code maintenance and evolution.**
   (a) To compile and analyze scholarly work on the application of DL methods to detect various types of code smells.
   (b) To identify and classify the datasets, code smells addressed, DL architectures used, and performance metrics reported in past research.
   (c) To evaluate gaps in existing approaches, such as generalizability, explainability, dataset imbalance, and lack of semantic understanding.
   (d) To suggest future research directions for using hybrid, interpretable, and scalable models for robust code smell detection.
2. **Investigate and utilize DL algorithms for the purpose of detecting software antipatterns.**

(a) **Addressing class imbalance in code smell datasets**

i. To investigate the impact of class imbalance on DL models for code smell detection.

ii. To evaluate and compare traditional resampling techniques and identify their limitations in code-based datasets.

iii. To propose and validate advanced resampling techniques that preserve semantic and syntactic consistency in source code representations.

iv. To examine the influence of these strategies on model stability, generalization, and minority class detection.

(b) **Improving code smell detection using ensemble DL**

i. To explore the suitability of CNNs, LSTMs, and GRUs in capturing both local and long-term patterns in source code.

ii. To design stacking-based ensemble models combining diverse DL architectures for improved code smell detection.

iii. To compare ensemble models with individual base learners across multiple code smells using standard metrics.

iv. To validate the effectiveness of ensembles through statistical tests confirming significant performance gains.

(c) **Exploring graph-based representations for improved code smell detection**

i. To construct a heterogeneous code graph that integrates both structural dependencies and semantic relationships within source code.

ii. To leverage GCNs and Graph Attention Network (GAT)s for deep representation learning from source code.

iii. To enable accurate multi-label classification of code smells such as Blob, Data Class, Feature Envy, and Long Method.

iv. To experimentally validate the proposed model on the textual dataset and compare it with existing state-of-the-art techniques.

v. To enhance the interpretability of the model through explainable graph-based learning mechanisms.

(d) **Investigating the role of semantic and structural code metrics in quantum software systems**

i. To establish a set of software metrics tailored for quantum programs by analyzing their structural and semantic characteristics.

ii. To utilize quantum circuit analysis and parse trees for extracting semantic insights from quantum code.

iii. To design rules and pattern-based strategies for detecting code smells in quantum software architecture.

iv. To preprocess and refine the quantum dataset into a labelled dataset of quantum programs annotated for code smells.

v. To validate the proposed quantum metrics and smell labels through expert reviews and performance classification.

vi. To evaluate the effectiveness of combined structural and semantic metrics in accurately identifying quantum code smells.

(e) **Developing a DL-based hybrid framework for detecting quantum code smells**

i. To investigate the applicability of DL methods for analyzing quantum code structures.

ii. To construct a labelled dataset of quantum programs annotated with different types of code smells.

iii. To build and evaluate a robust DL model capable of identifying various quantum code smells effectively.

iv. To compare the proposed model with baseline and transfer learning approaches to assess detection performance.

v. To understand the potential impact of quantum code smells on program quality and system performance.

vi. To demonstrate the advantages of hybrid neural network models in learning complex representations in quantum code.

3. **Explore and exploit the use of evolutionary algorithms in order to optimize the antipattern detection process.**

(a) To explore the importance of feature selection in enhancing the performance of code smell detection models.

(b) To reduce model complexity and improve classification accuracy by identifying the most relevant features.

(c) To integrate DL-based feature extraction with a robust feature selection mechanism.

(d) To develop an evolutionary algorithm that can overcome local optima and ensure effective feature subset optimization.

(e) To evaluate the effectiveness of the proposed framework in improving detection accuracy and computational efficiency.

(f) To highlight the potential of hybrid approaches combining DL and quantum-inspired optimization techniques.

4. **Evaluate the performance and determine the effectiveness of benchmark detection and correction tools used for software antipatterns.**

   (a) To explore the need for specialized software analysis tools tailored to the quantum programming paradigm.

   (b) To examine the capabilities of existing tools in identifying common types of errors and anomalies in quantum code.

   (c) To apply a standardized benchmark dataset for consistent evaluation across tools.

   (d) To compare multiple tools based on performance metrics such as accuracy, efficiency, and usability.

   (e) To identify the strengths and limitations of each tool in real-world quantum software scenarios.

   (f) To provide insights into the current state of bug detection in quantum programming and suggest directions for future tool development.

1.7 Overview of the Work

Code smell detection plays a pivotal role in ensuring software quality, maintainability, and long-term sustainability, both in classical and quantum computing paradigms. With the growing complexity of modern software systems, detecting structural and semantic anomalies, commonly referred to as *code smells*, has become essential to prevent future maintenance issues and technical debt. The overarching aim of this research is to enhance code smell detection mechanisms by addressing key challenges such as data imbalance, feature redundancy, model interpretability, and the absence of robust tools in quantum software analysis. This section presents a comprehensive overview of the sequential body of work undertaken, highlighting the motivation, methodology, and major contributions of each phase.

The journey began with a **SLR** following Kitchenham's guidelines to consolidate existing knowledge in code smell detection across classical domain. The SLR revealed substantial gaps, including limited representation of minority smells, lack of explainability in models, and the need for quantum-specific detection frameworks.

To address data-level challenges, particularly class imbalance in smell datasets, the **DIR-SMOTE** algorithm is proposed. This novel oversampling technique integrates density and influence-based mechanisms to generate more semantically relevant synthetic samples, thereby improving the performance of machine learning classifiers on minority smell classes. Building upon balanced datasets, the research explored advanced ensemble DL architectures such as CNN-GRU-LSTM, which effectively modeled sequential and spatial features of code snippets. Although these models improved detection rates, they fell short in capturing complex structural relationships inherent in source code. This led to the development of **SmellGraph**, a GNN-based approach that represents each class as a multi-view Code Property Graphs (CPG) combining its AST, Control Flow Graph (CFG), Program Dependence Graph (PDG) and inter-class dependencies, and learns over it using a hierarchical relational graph attention network guided by design metrics. SmellGraph demonstrated superior performance over existing approaches in detecting multiple code smells, both within and across projects, while providing interpretable, method-level explanations for its predictions. To extend this work into the quantum software domain, a set of domain-specific code metrics was designed and validated under the framework named **QUAMET**. These metrics, derived from the LintQ dataset of Qiskit-based programs, addressed the unique operational and architectural aspects of QC such as quantum depth, entanglement, and hybrid execution. This quantum exploration laid the foundation for building **QNET**, a DL-based model designed to detect code smells in quantum programs. QNET is trained using the curated quantum smell dataset and achieved promising results in smell identification using CNN-GRU-LSTM architectures, marking one of the earliest attempts at automated quantum code quality assessment.

Recognizing the need for optimal feature selection and reduction of metric dimensionality, a **Modified Quantum-Inspired Evolutionary Algorithm (*m-QIEA*)** is proposed. This approach effectively selected compact yet informative subsets of features for both classical (*MLCQ*) and quantum (*LintQ*) datasets, enhancing classification accuracy and model interpretability while balancing exploration and exploitation.

The research culminated in a practical evaluation of the current tooling ecosystem for quantum code quality analysis. A comparative study of tools such as **QChecker**, **QSmell**, and **QPAC** was conducted using the Bugs4Q benchmark suite, analyzing their effectiveness across real-world buggy quantum programs.

This work concludes with a critical discussion on the limitations encountered during the study, including potential threats to validity, dataset constraints, and evolving standards in quantum software development. The findings underscore the broader social impact of enhancing software quality, both in classical and quantum domains, by enabling more maintainable, efficient, and error-resilient systems that can reduce tech-

nical debt and improve developer productivity. Finally, future guidelines are provided to facilitate further exploration in this emerging field, including directions for refining quantum-specific metrics, enhancing cross-platform tooling, benchmarking against larger and diverse datasets, and fostering industry adoption of automated smell detection in next-generation software development environments.

1.8 Organization of Thesis

This thesis is organized into ten chapters, each building upon the previous to comprehensively address the problem of code smell detection using emerging technologies. The opening **Chapter 1** introduces the research context, motivation, and objectives of the study. **Chapter 2** outlines the research methodology adopted throughout the work. **Chapter 3** presents a systematic literature review to identify existing gaps and challenges in classical and emerging approaches. **Chapter 4** explores data-related issues and proposes solutions for handling imbalanced and complex datasets. **Chapter 5** focuses on the application of DL and graph-based models for improved detection. **Chapter 6** presents a quantum-inspired framework based on structural and semantic code metrics for enhancing code smell detection. **Chapter 7** introduces a hybrid learning approach combining classical deep models with quantum components to improve prediction accuracy and model expressiveness. **Chapter 8** explores evolutionary optimization strategies for refining model performance. **Chapter 9** evaluates and compares available tools and frameworks within the domain. The final **Chapter 10** summarizes the key findings, outlines the societal relevance, and provides directions for future research. The brief description of each chapter is given below.

Chapter 1 : This chapter introduces antipatterns in software engineering, highlighting their impact on code quality and maintainability. It outlines the importance of automated antipattern detection in the context of modern, complex software systems and emphasizes key challenges such as class imbalance, feature dimensionality, generalization across projects, and lack of model interpretability. The evolution of detection approaches is discussed, from rule-based methods to ML and AutoML solutions. The motivation, scope, and significance of the research are presented, followed by a formal research problem statement, research questions, and corresponding objectives that guide the direction of this study.

Chapter 2: This chapter outlines the comprehensive research methodology adopted to investigate the predictability of antipatterns using ML techniques. It begins by describing the overall research process and the formulation of the problem, followed by a detailed literature survey to establish the foundation and identify key gaps. The variables involved

in the study, both independent and dependent, are clearly defined to ensure a structured analytical framework. The chapter then presents various data analysis methods, including traditional ML classifiers, DL models, evolutionary algorithms, and AutoML tools, emphasizing their suitability for software quality prediction. An extensive experimental design is detailed, covering empirical data collection from real-world software systems, preprocessing steps, and advanced data balancing techniques to address class imbalance. Finally, the development and validation of prediction models are discussed, along with the evaluation metrics and statistical tools used to measure performance, ensuring the reliability and robustness of the proposed solutions.

Chapter 3: This chapter presents a SLR that critically examines the application of DL in code smell detection. Covering 35 primary studies selected from 8,739 publications between 2013 and the present, this work investigates the evolving landscape of DL-based approaches for identifying software design flaws. It identifies commonly addressed code smells such as Feature Envy, God Classes, Long Methods, Complex Classes, and Large Classes. The analysis reveals that models built using Recurrent Neural Network (RNN)s and CNNs, either individually or in hybrid settings, are most frequently adopted due to their robust learning capabilities. Despite the promising results, the study exposes significant research gaps, particularly related to subjectivity in smell definitions, inconsistent detection practices, and challenges in threshold tuning. Additionally, the study outlines future directions, including improvements in metric selection and dataset diversity, thereby laying a solid foundation for advancing the state of DL-driven code smell detection.

Chapter 4: This chapter addresses a critical challenge in code smell detection: class imbalance between smelly and non-smelly code instances. Traditional oversampling techniques such as SMOTE, Borderline-SMOTE, and Adaptive Synthetic Sampling (ADASYN) often fail to preserve semantic consistency, leading to noisy or irrelevant synthetic data. To overcome these limitations, this study introduces Density and Influence-based Resampling using SMOTE (DIR-SMOTE), a novel oversampling strategy that combines local density estimation with SHAP-based feature importance to generate high-quality, interpretable synthetic instances. DIR-SMOTE first eliminates noisy samples based on density metrics, then selectively perturbs top-ranked SHAP features during sample generation to preserve semantic meaning. Evaluated on five benchmark datasets, namely, Apache, jEdit, EDTForCSD, DesigniteJava, and MLCQ, DIR-SMOTE consistently outperforms nine standard resampling methods, achieving improvements of up to 6.7% in F1-score and 5.1% in precision. It proves effective in both binary and multiclass settings and offers practical advantages in terms of computational efficiency and interpretability. This chapter demonstrates DIR-SMOTE's potential as a scalable and domain-aware solution

for addressing data imbalance in software quality assurance.

Chapter 5: This chapter presents two complementary DL approaches for code smell detection. The first work investigates ensemble DL models that combine the strengths of individual architectures, using CNNs to extract local features and LSTM and GRU models to capture sequential dependencies. Both homogeneous and heterogeneous ensembles are developed, using soft voting to improve predictive robustness. Evaluated on five code smells, namely Large Class, Refused Bequest, Complex Class, Lazy Class and Spaghetti Code, the ensembles consistently outperform individual DL models, as confirmed by statistical validation. While the ensembles improve detection, they still treat source code as metric vectors or token sequences, and therefore overlook the structural relationships, such as control flow, data dependence, inheritance and invocation, through which most smells manifest. To address this limitation, the second work proposes **SmellGraph**, a GNN-based approach that represents each class as a multi-view CPG combining its AST, CFG, PDG and inter-class dependencies, and learns over it using a hierarchical relational graph attention network guided by design metrics. SmellGraph outperforms existing approaches both within and across projects while providing interpretable, method-level explanations for its predictions. Together, these works show how progressively richer code representations lead to more accurate and reliable code smell detection.

Chapter 6: This chapter presents **QUAMET**, a novel approach for identifying code smells in Quantum Program (QP)s by combining both *structural* and *semantic* analysis. With the rapid advancement of QC and the introduction of new programming paradigms, the likelihood of code smells and bugs in QPs is growing. However, limited research exists in detecting these issues specific to quantum software. QUAMET addresses this gap by establishing a set of **domain-specific software metrics** tailored for quantum programming. Structural analysis is used to capture architectural and design-level patterns, while semantic analysis leverages quantum circuit insights and parse trees to interpret code behavior. The study utilizes the **LintQ dataset**, containing real-world Qiskit-based QPs, and extracts a refined dataset of 4,102 labeled projects. This dataset supports the detection of six prominent quantum code smells, validated through expert feedback from academia and industry. The chapter demonstrates that combining structural and semantic insights significantly enhances the detection of design flaws in QPs, with implications for improving software **performance, maintainability**, and overall **quality** in QC systems.

Chapter 7: This chapter investigates the novel application of DL techniques for the detection of code smells in QPs, a critical yet underexplored area in the field of QC. Recognizing that quantum code smells can significantly impair the maintainability and performance of quantum systems, this work presents a comparative ablation study of various DL models, namely CNN, GRU, and LSTM networks. A hybrid model integrating all

three architectures is proposed and benchmarked against traditional models. To support this study, a manually curated and labeled dataset comprising 136 open-source QPs was created by extracting quantum circuits. The hybrid model demonstrates superior accuracy (92.86%) compared to individual DL models and a transfer learning baseline using Bidirectional Encoder Representations from Transformers (BERT). This chapter not only establishes the feasibility of DL-based quantum code smell detection but also highlights key research questions related to the impact of quantum code smell on quantum system reliability and the complementary strengths of diverse DL architectures. The findings emphasize the promise of DL in addressing code quality challenges specific to quantum software development.

Chapter 8: This chapter presents a Feature Selection (FS)-driven framework for enhancing code smell detection in both classical and quantum software systems. Recognizing the limitations of traditional ML approaches, such as feature redundancy, increased computational cost, and suboptimal model performance, this work proposes a hybrid solution that integrates DL-based feature extraction with a modified Quantum-Inspired Evolutionary Algorithm (m-QIEA) for refined feature selection. The m-QIEA algorithm is designed to avoid local optima stagnation, thereby ensuring a more robust and optimized selection of features. The model is evaluated on both classical and quantum datasets, demonstrating consistent improvements in classification accuracy and computational efficiency. Comparative studies with traditional FS methods and state-of-the-art Large Language Model (LLM)-based smell detectors further validate the superiority of m-QIEA. The results underscore the importance of effective FS in maximizing detection performance while minimizing resource utilization, establishing the proposed framework as a scalable and generalizable solution for cross-paradigm code smell detection.

Chapter 9: This chapter presents a comparative study of existing tools designed to detect software bugs in QPs. As QC matures, ensuring code quality becomes increasingly critical due to the unique architectural and logical complexities inherent in quantum software. The chapter evaluates three widely used bug detection tools, namely, QChecker, QSmell, and QPAC, using the Bugs4Q benchmark dataset, which contains 36 quantum code samples with varying types of bugs. Each tool is assessed for its detection capability, efficiency, and usability after standardized pre-processing of the dataset. The comparative analysis reveals that QChecker consistently outperforms the others in terms of detection accuracy and tool effectiveness, making it a reliable choice for identifying bugs in QPs. The insights derived from this analysis guide developers and researchers toward selecting appropriate tools for quality assurance in quantum software development.

Chapter 10: This final chapter presents a comprehensive summary of the research work conducted on antipattern detection using ML techniques. It recaps the major find-

ings and key contributions, emphasizing how the proposed models address class imbalance, improve FS, and enhance interpretability and generalizability. The chapter also outlines limitations and proposes future research directions, including real-time detection, integration with Continuous Integration / Continuous Deployment (CI/CD) pipelines, and the use of quantum and edge-based solutions for scalable deployment. Finally, the broader social implications are discussed, such as supporting better software maintainability, reducing technical debt, and promoting reliable software systems, thereby benefiting both developers and end-users.

Chapter 2

Research Methodology

The experimental methodology for analyzing antipatterns and software maintainability follows a systematic and well-defined approach to ensure reliable empirical experimentation. This chapter outlines the methodology used in this research. The chapter is organized as follows: Section 2.1 explains the research process followed throughout the study. Section 2.2 defines the research problem addressed. Section 2.3 provides a comprehensive literature survey, outlining existing research in this domain. Section 2.4 defines the independent and dependent variables used in the study. Section 2.5 discusses the various data analysis methods employed, including ML, DL, evolutionary algorithms, and AutoML. Finally, Section 2.6 presents the experimental design framework, detailing the data collection, preprocessing, balancing techniques, prediction model validation, performance metrics, and statistical analysis used in the study.

2.1 Research Process

The research process is a structured and methodical sequence of steps designed to explore and address the research problem effectively. Figure 2.1 illustrates the research process followed throughout the chapters of this thesis. Each stage of the research process is elaborated below:

1. **Problem Definition:** The initial stage of the research process entails a precise and comprehensive definition of the problem under investigation. This involves outlining the scope of the study, formulating specific objectives, and identifying the key questions that the research seeks to address. A clearly defined problem statement serves as the foundation for all subsequent stages, ensuring focus and relevance.

2. **Review of Literature:** A systematic review of scholarly literature is conducted to examine existing theories, methodologies, and empirical findings related to the iden-

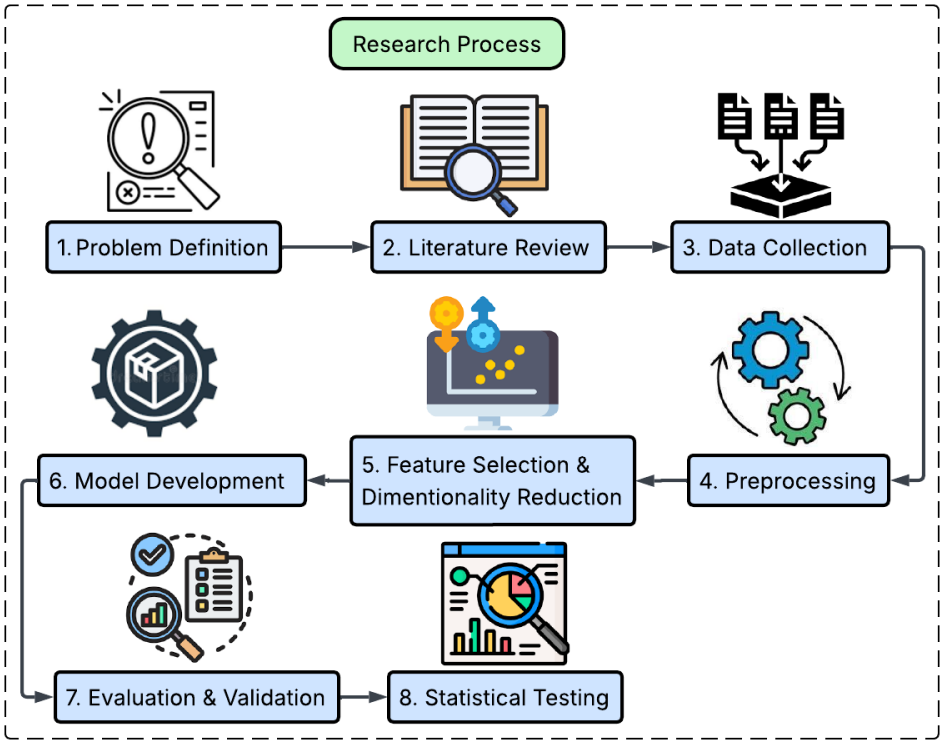


Figure 2.1: Research Process

tified problem. This step assists in recognising research gaps, understanding prior work, and refining the scope of the study. It also helps to position the present research within the broader academic discourse.

3. **Data Collection:** Data relevant to the research objectives is acquired from appropriate and reliable sources, which may include experimental setups, publicly available datasets, institutional repositories, or surveys. The selection of data sources is guided by the need for accuracy, completeness, and representativeness.

4. **Data Preprocessing and Balancing:** Prior to analysis, the collected data undergoes a preprocessing phase to remove inconsistencies, handle missing values, and ensure uniformity of format. Where applicable, class imbalance issues are addressed through balancing techniques such as oversampling, undersampling, or synthetic data generation, thereby improving the quality of the subsequent analysis.

5. **Feature Selection:** To enhance the efficiency and interpretability of the analysis, relevant features that significantly influence the outcome are identified using feature selection techniques.

6. **Model Development:** Appropriate computational or statistical models are developed or adapted to address the defined problem. This involves training the models on the prepared dataset, selecting suitable parameters, and applying relevant algorithms to produce robust and generalizable solutions.

7. **Evaluation and Validation:** The developed models are subjected to a rigorous evaluation process to assess their accuracy, reliability, and predictive capability. This includes the application of relevant performance metrics and the use of validation techniques such as cross-validation or external test sets to ensure the generalizability of the results.

8. **Statistical Testing:** To confirm the robustness and significance of the findings, statistical tests are applied. These tests help determine whether the observed results are attributable to genuine effects rather than random variation, thereby reinforcing the credibility of the research outcomes.

2.2 Defining Research Problem

The first and most critical step in the research process is to define the research problem with clarity and precision. This involves identifying the specific issues to be addressed, formulating them as Research Question (RQ)s, and aligning them with the broader objectives of the study. In the context of this thesis, the problem revolves around the challenges of detecting code smells in software systems, particularly when dealing with imbalanced datasets, and exploring advanced approaches to enhance detection accuracy and interpretability. This research seeks to answer the following RQs:

1. What is the current state of literature in the domain of code smell detection, particularly concerning imbalanced datasets, and what research gaps remain unaddressed?

2. What is the impact of dataset imbalance on the performance of code smell prediction models, and how does it affect their generalizability across projects?

3. Which advanced sampling techniques (classical and modern) can be employed to mitigate class imbalance in code smell detection datasets effectively?

4. How do different ML and DL approaches, including ensemble and graph-based models, compare in terms of detection accuracy, robustness, and interpretability?

5. Which evolutionary algorithms can be adapted or enhanced to perform optimal FS in the context of code smell detection?

6. How can the performance of the proposed framework be improved through the integration of AutoML tools, and which configurations yield the most effective results?

7. How do different software tools, libraries, and frameworks contribute to enhancing the efficiency and scalability of the proposed methodology?

8. What experimental design framework including data collection, preprocessing, model validation, and performance metrics ensures the reliability and accuracy of the study's findings?

2.3 Literature Survey

The literature survey phase involves an extensive review of existing research works, scholarly articles, technical reports, and case studies relevant to the research domain. The primary aim is to gain a thorough understanding of the theoretical foundations, methodologies, datasets, and experimental results that have been reported in previous studies. This stage provides insights into the current state-of-the-art approaches and helps identify research gaps that the present study aims to address.

2.3.1 Steps to Conduct Literature Survey

i. **Define the Scope of the Survey**

- Clearly identify the central research questions and themes related to code smell detection, such as improving detection accuracy, handling imbalanced datasets, and integrating DL techniques.
- Restrict the scope to relevant time range, methodologies in detecting code smells, and specific challenges that can be identified

ii. **Source Identification**

- Identify credible sources including peer-reviewed journals, conference proceedings, technical reports, and open-source datasets such as MLCQ, EDTforCSD, Bug4Q, or project-specific repositories.
- Use reputable databases like IEEE Xplore, SpringerLink, ACM Digital Library, and Google Scholar for literature gathering.

iii. **Keyword Search** Develop a focused set of keywords, for example: "code smell detection," "machine learning for code smells," "quantum code smells," "code smell

class imbalance," "tools for code smell," and "evolutionary feature selection for smells."

iv. **Organize and Filter Results**

- Filter results based on relevance, citation count, recency, and alignment with the research problem. Prioritise studies presenting novel algorithms, innovative metrics, or advanced preprocessing approaches.
- Use citation management tools like Mendeley or EndNote for efficient organization.

v. **Critical Evaluation of Sources**

- Examine methodologies, datasets, and evaluation frameworks used in prior work.
- Pay particular attention to studies applying ML and DL techniques, graph-based models, or quantum-hybrid approaches, and assess the strengths and weaknesses in their experimental setups.
- Review the performance metrics (e.g., Accuracy, F1-score, MCC, AUC) and statistical validation techniques used.

vi. **Synthesizing Information**

- Combine insights from multiple studies to highlight recurring challenges and effective methodologies .
- Organize findings into thematic categories such as "Data Preprocessing and Balancing Techniques," "Deep Learning Approaches," "Evolutionary Algorithms for Feature Selection," and "Tool Support for Code Smell Detection."

vii. **Identify Research Gaps and Opportunities**
Identify areas where further improvement is needed, such as automated smell graph generation, advanced evolutionary feature selection methods, hybrid deep-quantum frameworks, or tool integration for real-time detection in CI/CD pipelines.

viii. **Writing the Literature Review**
Present the literature review in well-structured thematic sections, each addressing a major challenge in code smell detection. Discuss prior work, highlight gaps, and establish the rationale for the approaches proposed in this thesis.

To understand the research problem of detecting antipatterns and code smells in software systems, literature survey of existing studies is essential. Through this review, we can assess the extent to which these issues have been explored and the methods proposed for their identification. Antipatterns are recurring design solutions that initially seem effective but lead to long-term negative consequences such as increased complexity, reduced maintainability, and performance degradation [17, 72, 73, 74, 75]. These patterns often reflect suboptimal design decisions that can cause structural problems in the software, making it more difficult to modify and extend over time. Researchers have proposed various methods for detecting antipatterns, including static analysis tools, rule-based systems, and pattern recognition approaches, aimed at identifying and addressing these issues before they adversely impact the system's quality.

Code smells, on the other hand, are localized issues in the code that suggest potential weaknesses in design or implementation. While not immediate errors, code smells highlight areas that may require refactoring to improve software maintainability, readability, and performance [21, 76, 77, 78]. These smells, such as large classes, long methods, duplicated code, feature envy, and improper commenting, can degrade the software's structure over time. Numerous studies have focused on detecting code smells through static analysis, using software metrics to automatically identify potential issues within the code [21, 76]. Detecting code smells early is crucial, as addressing them promptly can prevent technical debt accumulation, reduce maintenance costs, and enhance the overall quality of the software. Several detection techniques have been developed, including machine learning-based methods, hybrid approaches, and pattern recognition algorithms, which assist developers in identifying and refactoring these smells efficiently.

2.4 Defining Variables

Variables are factors or elements in an experiment or study that can vary or be altered. In any model, the variables define the different elements that the model analyzes or predicts. These variables are generally categorized into independent and dependent variables, where independent variables serve as inputs to the model, and dependent variables represent the outputs that the model aims to predict or classify. A clear understanding of these variables is essential for developing accurate models capable of generalizing across diverse software systems.

2.4.1 Independent Variables

Independent variables are the features or attributes used by the ML model to make predictions or classifications. In the context of code smell detection, the independent variables are the metrics or measurements extracted from the codebase that potentially highlight the presence of code smells. These variables are classified into *Classical Metrics* and *Quantum Metrics* depending on the type of software being analyzed.

2.4.1.1 Classical Metrics

Classical metrics are those typically used in traditional software engineering to evaluate and detect smells in code. These metrics aim to assess the quality of the code based on factors like complexity, structure, and readability, which can affect software maintenance and overall performance. Below are some key classical metrics that are relevant for code smell detection:

- **Cyclomatic Complexity (CYCLO)**: This metric measures the number of linearly independent paths in a program's source code. Higher complexity typically indicates a more convoluted and harder-to-maintain codebase.
- **Code Churn (CHURN)**: Refers to the percentage of lines of code added, modified, or deleted over time. Excessive code churn can be a sign of unstable or poorly maintained code.
- **Coupling Between Objects (CBO)**: It is a measure of the interdependence between classes. Higher coupling often indicates a higher likelihood of code smells like tight coupling, which makes the system harder to maintain and test.
- **Depth of Inheritance Tree (DIT)**: This metric measures the maximum length from a given class to the root of the inheritance hierarchy. A high DIT can lead to problems such as excessive coupling or complexity.
- **Number of Methods per Class (NOM)**: This metric counts the number of methods within a class. A high number of methods can signal large classes, which might be harder to maintain and prone to the "God Class" code smell.
- **Lines of Code (LOC)**: This is a measure of the size of the codebase. Large classes or methods with excessive LOC can lead to the "Long Method" or "Large Class" code smells.

- **Fan-In Metric (FANIN) and Fan-Out Metric (FANOUT)**: These metrics measure the number of functions or classes calling a specific method or class (fan-out) and the number of methods or classes that are called by the method or class (fan-in). High values indicate tightly coupled components, often leading to undesirable code smells.
- **Code Duplication (CDUP)**: It is a measure of duplicated code fragments within the codebase. High duplication can lead to maintenance issues.
- **Comment Density (CDEN)**: This measures the ratio of comments to LOC. A low comment density could indicate that the code is poorly documented, which often correlates with the "Insufficient Documentation" smell.
- **Weighted Methods per Class (WMC)**: This metric sums the complexities of all methods in a class, often using CYCLO as the base measure. A higher WMC indicates that the class is performing multiple complex operations, making it more difficult to maintain, test, and understand.
- **Tight Class Cohesion (TCC)**: It measures the degree to which methods in a class are related through shared attributes. A low TCC value suggests poor cohesion, often indicating that the class is performing unrelated tasks.
- **Maximum Nesting Level (MAXNESTING)**: This metric captures the deepest level of nested control structures within a method. Excessive nesting often reduces code readability and can increase the risk of errors, making maintenance more challenging.
- **Number of Accessed Variables (NOAV)**: Counts the number of getter and setter methods in a class. A high NOAV may indicate excessive exposure of internal data, leading to issues with encapsulation.
- **Lack of Cohesion of Methods (LCOM)**: Measures the degree to which methods in a class share common instance variables. Higher LCOM values suggest poor cohesion, meaning the class may be handling multiple unrelated responsibilities.
- **Number of Added Methods (NOAM)**: Represents the total count of methods introduced in a given class over a specific version or time frame. High NOAM may point to feature bloat or evolving design without adequate refactoring.
- **Coupling Intensity (CINT)**: Measures the total number of distinct external classes a given class interacts with. Higher suggests stronger coupling, which can reduce modularity and increase the impact of changes.

- **Coupling Dispersion (CDISP)**: It indicates the distribution of a class's methods and dependencies across various packages. A high dispersion value may lead to maintenance difficulties due to scattered dependencies.
- **Access to Foreign Data (ATFD)**: Counts how many times a class accesses attributes from other classes. High ATFD may indicate feature envy or violation of encapsulation principles.
- **Locality of Attribute Accesses (LAA)**: Represents the proportion of a class's attribute accesses that are internal to the class. A low LAA suggests frequent reliance on external data, which can be a design weakness.
- **Foreign Data Providers (FDP)**: Measures the number of distinct classes from which a given class accesses attributes. A high FDP count suggests strong dependencies on other classes' internal structures.
- **Number of Parameters (PAR)**: It refers to the number of parameters a method accepts. Excessively long parameter lists can reduce readability and increase the likelihood of the "Long Parameter List" smell.
- **Base Class Usage Ratio (BUR)**: Indicates the extent to which methods and attributes inherited from base classes are used in the subclass. A low BUR may suggest unnecessary or misapplied inheritance.
- **Number of Parents (NOP)**: It measures how many direct superclasses (parent classes) a given class inherits from. A high NOP value indicates complex inheritance hierarchies, which may increase complexity and make maintenance harder.
- **Change Metric Spread (CM)**: It measures how widely a change to a single functionality spreads across classes.
- **Static Elements Count (SEC)**: It counts the number of static methods and attributes within a class. Overuse of static elements can reduce flexibility and hinder testability.
- **Static Data Complexity (SDC)**: Represents the degree of complexity introduced by static data structures and variables. Higher SDC may indicate shared state problems and reduced thread safety.
- **Lack of Balance (LB)**: It evaluates the distribution of responsibilities among classes or methods. Poor balance can lead to uneven workloads, resulting in overly complex or underutilized components.

- **Number of Children (NOC)**: Counts the number of immediate subclasses of a given class. A high NOC can be a sign of over-generalisation or excessive subclassing.
- **Average Method Weight (AMW)**: It calculates the average complexity of methods in a class. Higher AMW values suggest that the class contains more computationally intensive methods.
- **Response for a Class (RFC)**: Calculates the number of distinct methods that can be invoked in response to a message sent to a class, including methods invoked by other methods. A high RFC increases testing complexity and may indicate excessive responsibility.

2.4.1.2 Quantum Metrics

Quantum metrics focus on the unique characteristics and performance indicators of quantum software. These metrics are particularly relevant when dealing with QC environments, as QPs often involve quantum gates, qubits, and other quantum-specific components. Below are key quantum metrics used for code smell detection in quantum software:

- **Number of Qubits**: This metric tracks the number of qubits used in a quantum program. A higher number of qubits can make the system more prone to noise and error, leading to performance issues or inefficient algorithms.
- **Number of Quantum Gates**: This measures how many quantum gates are used in the quantum program. Excessive use of quantum gates might lead to increased complexity and inefficiency in the quantum circuit, potentially leading to resource-related code smells.
- **Quantum Circuit Depth**: This refers to the number of layers or time steps in a quantum circuit. A deeper circuit generally indicates greater complexity, which could increase the chances of errors or inefficiencies.
- **Clbits (Classical Bits)**: Clbits are used to store classical data in quantum algorithms. A high number of clbits may indicate that too much classical processing is being integrated into the quantum algorithm, which could lead to inefficiencies.
- **Gate Count per Qubit**: This metric tracks how many gates are applied to each qubit in the quantum circuit. High gate counts per qubit could suggest inefficiencies in the quantum algorithm, possibly leading to longer execution times or higher error rates.

- **Quantum Entanglement**: This measures the level of entanglement within a quantum program. While entanglement is necessary for many quantum operations, excessive or unnecessary entanglement can lead to complex and inefficient programs that are prone to errors.
- **Quantum Error Rate**: This tracks the frequency of errors in the quantum operations. A high error rate is often an indicator of problems with the quantum algorithm's robustness or implementation.
- **Quantum Circuit Optimizations**: Metrics related to the optimization of quantum circuits, such as gate reduction or simplification of quantum operations, are critical. Poor optimization can lead to inefficiency, longer execution times, and increased resource usage.
- **Quantum Resource Usage**: This metric evaluates the consumption of quantum resources such as qubits, gates, and time steps during the execution of the program. Excessive resource usage could indicate suboptimal code that could benefit from refactoring.

A few of the other quantum metrics are described in Section 6.3.2.

2.4.2 Dependent Variables

In code smell detection, *dependent variables* refer to the target outputs that the models aim to predict. They represent the presence or absence of specific types of code smells, and are directly influenced by the independent variables (features) extracted from the code itself. Code smells are indicators of suboptimal code structure that can negatively impact software maintainability, performance, or readability. The nature of the dependent variables changes based on whether the problem is posed as a binary classification or multiclass classification task.

There are two key categories of smells being studied: *classical smells* (pertaining to traditional software engineering) and *quantum smells* (relevant to quantum software development). Each category has its own binary and multiclass classification structures.

2.4.2.1 Classical Smells: Binary and Multiclass

Classical smells are the well-established software design flaws that developers and researchers have traditionally focused on. These smells are symptoms of deeper structural problems in the e code, such as violating object-oriented design principles, creating overly complex or rigid code structures, or making the codebase harder to maintain and extend.

Binary Classification of Classical Smells In a *binary classification* context, the dependent variable indicates whether a specific classical code smell is present or absent in a given piece of software. The model output is a simple yes/no prediction. For example:

- 1 = The code contains a specific classical smell.
- 0 = The code does not contain that smell.

This binary approach is used when detecting a single type of smell or determining whether any smell exists without distinguishing between different types.

Multiclass Classification of Classical Smells In the context of multiclass classification, a model is tasked not only with identifying the presence of a smell but also categorizing the specific type of smell affecting the system. This enables a more nuanced understanding of software quality and helps prioritize refactoring efforts. Table 2.1 defines a short description of classical code smells, followed by the detailed explanation of the most common smells, along with their identification criteria and key indicators of severity.

Table 2.1: Classification of software antipatterns along with their identification criteria and severity indicators.

Antipattern	Description	Identification Criteria	Severity Indicator
Class-Level Antipatterns			
Blob	Large class handling excessive responsibilities.	Many methods and fields with low cohesion.	Difficult to understand, maintain, and test.
Brain Class	Class combining multiple unrelated responsibilities.	Violation of the Single Responsibility Principle.	Complicated maintenance and limited extensibility.
God Class	Single class controlling major system functionality.	Manages numerous objects and responsibilities.	Becomes a maintenance bottleneck.
Large Class	Class containing excessive attributes and methods.	Class size significantly larger than average.	Reduced maintainability and readability.
Lazy Class	Class providing minimal functionality.	Small and underutilized implementation.	Adds unnecessary complexity.

Continued from previous page

Antipattern	Description	Identification Criteria	Severity Indicator
Data Class	Class containing mostly data with little behaviour.	Primarily fields and getter/setter methods.	Weak encapsulation and scattered business logic.
Method-Level Antipatterns			
Brain Method	Overly long and complex method.	Many variables, branches, and nested logic.	Difficult testing and modification.
Long Method	Method containing excessive implementation logic.	High line count and complexity.	Poor readability and maintainability.
Long Parameter List	Method requiring numerous parameters.	Large number of input arguments.	Error-prone and difficult to understand.
Feature Envy	Method relying heavily on external classes.	Accesses more foreign data than local data.	Indicates misplaced functionality.
Design and Inheritance Antipatterns			
Alternative Classes with Different Interfaces	Classes providing similar functionality through incompatible interfaces.	Similar behaviour but different method signatures.	Limits interchangeability and increases complexity.
Class Hierarchy Problem	Deep or overly complex inheritance structures.	Multiple inheritance levels and overrides.	Fragile architecture and difficult maintenance.
Parallel Inheritance Hierarchies	Multiple hierarchies evolving together.	Corresponding subclasses across hierarchies.	Duplication and strong coupling.
Refused Bequest	Subclass rejects inherited behaviour.	Unused or overridden inherited methods.	Violates inheritance principles.
Functional Decomposition	Object-oriented system decomposed into procedural units.	Many small classes with minimal state.	Loss of OO benefits and over-complexity.

Continued from previous page

Antipattern	Description	Identification Criteria	Severity Indicator
Switch Statements	Conditional logic replacing polymorphism.	Large switch/case structures.	Rigid design and duplicated logic.
Speculative Generality	Unused abstractions added for future needs.	Unused interfaces and extension points.	Unnecessary complexity.
Coupling and Communication Antipatterns			
Inappropriate Intimacy	Classes excessively dependent on each other's internals.	Frequent access to private implementation details.	Reduced modularity and increased fragility.
Insider Trading	Modules sharing internal data excessively.	Direct modification of another module's state.	Violates encapsulation.
Middle Man	Class primarily delegates work to others.	Methods mostly forward calls.	Introduces unnecessary indirection.
Message Chains	Long chains of object navigation.	Frequent nested accessor calls.	Creates tight coupling.
Shotgun Surgery	Single change requiring modifications in many places.	Multiple files changed for one feature.	High maintenance effort.
Divergent Change	Different changes affecting the same class repeatedly.	Class modified for unrelated reasons.	Poor separation of concerns.
Data-Related Antipatterns			
Data Clumps	Groups of data frequently occurring together.	Repeated parameter or field combinations.	Suggests missing abstraction.
Global Data	Data accessible from multiple system locations.	Global variables or shared mutable state.	Unpredictable side effects.

Continued from previous page

Antipattern	Description	Identification Criteria	Severity Indicator
Mutable Data	Data modified after creation.	Public mutable fields or weak immutability.	Unexpected behaviour and bugs.
Primitive Obsession	Overuse of primitive data types.	Complex concepts represented using primitives.	Poor expressiveness and scattered validation.
Temporary Fields	Fields used only in specific contexts.	Variables frequently null or conditionally assigned.	Confusing object state.
Maintainability and Readability Antipatterns			
Duplicate Code	Repeated implementation fragments.	Identical or near-identical code blocks.	Increased maintenance effort.
Comments	Excessive comments compensating for poor design.	Large amount of explanatory comments.	Indicates unclear code structure.
Mysterious Name	Unclear or misleading identifiers.	Vague names such as temp or foo.	Reduces understandability.
Spaghetti Code	Unstructured and tangled implementation.	Excessive nesting and poor modularization.	High complexity and low maintainability.
Lazy Element	Component providing negligible value.	Unused or trivial modules.	Unnecessary codebase clutter.
Incomplete Library Class	Library class lacking required functionality.	Reliance on limited third-party APIs.	Workarounds and unstable implementations.

In this setting, the model identifies not just whether the code is problematic, but also the nature of the problem. This is particularly useful when the goal is to prioritize different types of smells for refactoring efforts or to understand the overall health of a software system.

2.4.2.2 Quantum Smells: Binary and Multiclass

As QC evolves, *quantum smells* are emerging as an analogous concept to classical code smells, but within the specialized domain of quantum programming. Quantum smells are inefficiencies or poor practices in quantum algorithms and quantum software that may hinder the performance or scalability of quantum computations. Given the nascent nature of quantum programming, identifying and addressing quantum smells is critical to building more efficient quantum systems.

Binary Classification of Quantum Smells In a *binary classification* context for quantum smells, the dependent variable is a simple yes/no determination of whether a quantum-specific smell exists. For example:

- 1 = The code contains a quantum smell.
- 0 = The code does not contain any quantum smell.

Binary classification for quantum smells might focus on detecting general issues like inefficient qubit usage, unnecessary quantum gates, or suboptimal circuit design. These problems can lead to increased quantum resource consumption, which is a critical factor given the current hardware limitations of quantum computers.

Multiclass Classification of Quantum Smells In quantum software development, *quantum smells* highlight inefficiencies or problematic patterns in quantum circuits. These smells can affect the performance, efficiency, and scalability of quantum algorithms, making their detection crucial for optimizing quantum systems. Below are descriptions of various quantum smells that can be used in a multiclass classification setting:

1. **Use of Customized Gates**: Introduces uniquely defined gates not in the standard backend library, which may impair portability.
    - *Identification Criteria*: Non-native, user-defined gates appear instead of standard ones.
    - *Key Indicator of Severity*: Greater depth or compilation difficulties on other hardware.

$$\mathrm{UCG} \;=\; \frac{|\{\, g \in G \mid \mathrm{type}(g) \notin \mathcal{B} \,\}|}{|G|}$$

Where G is the multiset of gates in the circuit, and $\mathcal{B}$ the backend's native gate set.

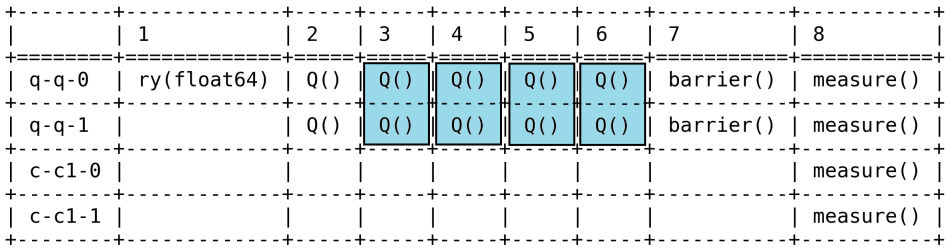


Figure 2.2: ROC- Test Amplitude Estimator circuit

2. **Repeated Set of Operations on Circuit**: Occurs when identical gate sequences are duplicated consecutively, rather than modularized.

   - *Identification Criteria*: Same subcircuit appears multiple times in a row.
   - *Key Indicator of Severity*: Increased noise and maintenance challenges.

   $$\mathrm{ROC} \;=\; \max_{L}\; \max_{i}\; \max_{k\geq 2}\; \mathbf{1}[\,\forall\, t \in \{0,\ldots,k|L|-1\},\; G[i+t] = L[t \bmod |L|]\,] \cdot k$$

   where G represents the quantum gate sequence, L is a repeated subsequence (loop) of gates, $|L|$ denotes the length of L, i is the starting index in G, k is the repetition count, and $\mathbf{1}[\cdot]$ is the indicator function returning 1 when the condition holds and 0 otherwise. This concept is illustrated in Figure 2.2, which showcases the "Test Amplitude Estimator" circuit. Notably, the critical observation in the circuit is that the initial operation sequence is replicated four times in an unvarying order. As a result, the ROC value for this scenario is 4.

3. **Non-parameterized Circuit**: Static circuits encoded without tunable inputs, limiting flexibility.

   - *Identification Criteria*: No symbolic parameters present in the circuit design.
   - *Key Indicator of Severity*: Cannot adapt or train; poor reuse potential.

   $$\mathrm{NPC} \;=\; \begin{cases} 1, & \text{if } |P| = 0 \\ 0, & \text{otherwise} \end{cases}$$

   Where P denotes the set of circuit parameters.

4. **Long Circuit**: A quantum circuit whose depth is excessive relative to device coherence time.

   - *Identification Criteria*: Longest dependency path (gate depth) is large.
   - *Key Indicator of Severity*: High chance of decoherence and noise.

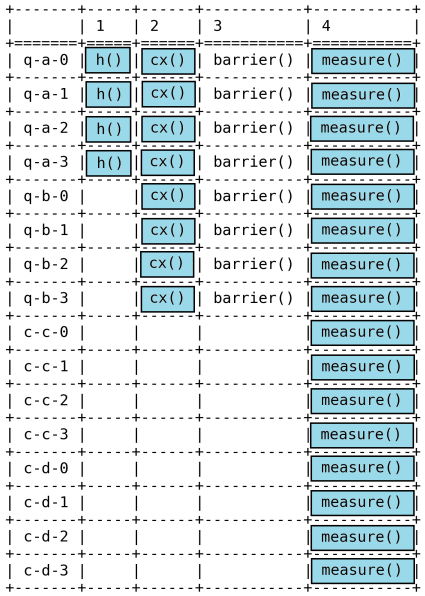


Figure 2.3: LC- Load Qasm circuit

$$\text{Depth} \;=\; \max_{\pi \in \mathcal{P}} \; |\pi|$$

Where $\mathcal{P}$ is the set of all topologically ordered paths in the circuit's gate dependency graph.

These circuits exhibit program and operational lengths that exceed a predefined threshold value. When a circuit displays considerable depth, i.e., a high number of operations, the error rate on the backend of the quantum circuit tends to rise. Simplifying these lengthy circuits into more concise ones can lead to substantial computational time savings. The figure illustrates a circuit labelled "Load Qasm," which contains a long circuit. As seen in Figure 2.3, the quantum circuit's length is notably extended. This is due to the concurrent utilisation of four quantum registers and four classical registers in three distinct operations, resulting in a combined circuit length of 28, which aligns with the LC metric value.

5. **Intermediate Measurements**: Measurements embedded in the middle of circuits, not only at the end.

   - *Identification Criteria*: Any measurement occurring before the final execution layer.
   - *Key Indicator of Severity*: Causes collapse and disrupts coherent quantum operations.

$$\text{IM} \;=\; \left| \left\{ m \in M \mid \ell(m) < \max_{g \in G} \ell(g) \right\} \right|$$

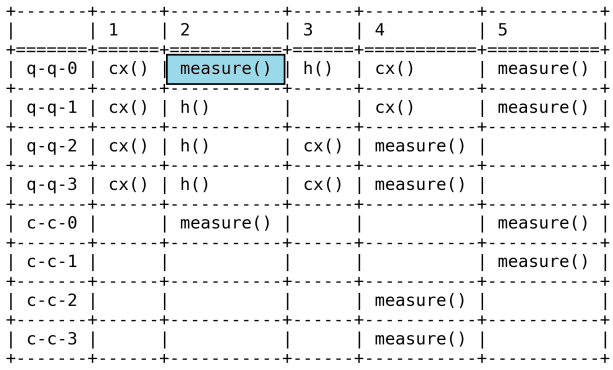


Figure 2.4: IM- Stochastic Swap circuit

Where G is the gate multiset, $M \subseteq G$ the measurements, and $\ell(\cdot)$ the layer index in a valid schedule. A measurement is "intermediate" if it does not occur on the last layer $\max_{g \in G} \ell(g)$. The set cardinality counts all such occurrences. The measure operation in QP extracts classical information from quantum bits. However, it's important to note that the quantum bit collapses after this operation and becomes unstable. The measure operation is typically strategically placed as the final step within a quantum circuit. This precaution is taken because performing measurements at intermediate stages can introduce errors to the circuit. However, this scenario is called Intermediate Measurement (IM) when the measurement function is inserted within the circuit. Therefore, adhering to the principle of deferred measurement, it is advisable to postpone all measurements until the final section of the quantum circuit.

As shown in Figure 2.4, the circuit "Stochastic Swap" features a measure operation occurring in the middle of the sequence. This departure from the deferred measurement principle results in an IM value of 1 for this circuit configuration.

6. **Idle Qubits**: Qubits left inactive across extended parts of the circuit.

   - *Identification Criteria*: Qubits with large gaps between their first and last operations.
   - *Key Indicator of Severity*: Vulnerable to decoherence and coherence-time waste.

$$\text{Idle} \;=\; \sum_{q \in Q} \sum_{j=1}^{r_q - 1} \max\Big(0,\; \ell_q(j+1) - \ell_q(j) - 1\Big)$$

where Q is the set of qubits, r_q denotes the total number of operations applied to qubit q, $\ell_q(j)$ is the index (time step) of the jth operation on qubit q, and the $\max(0, \cdot)$ term ensures that only positive idle gaps (i.e., consecutive time steps with no operations) are counted.

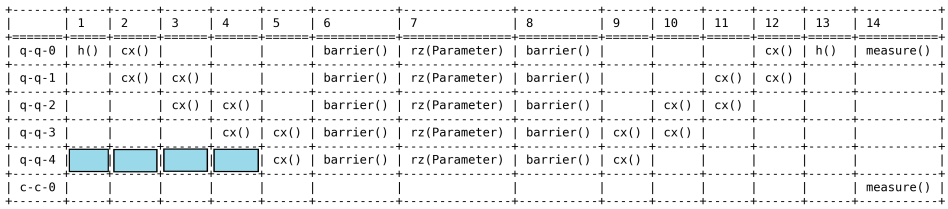


Figure 2.5: IdQ- Parametrized circuit

In the context of quantum coding, it is a common practice to initialise all qubits at the beginning. However, some qubits are intentionally left without active operations for a certain period. During this period of inactivity, the qubits tend to lose their stability. To restore their strength, a separate set of functions is applied. This state of the quantum circuit is referred to as Idle Qubit (IdQ). The circuit depicted in Figure 2.5 exemplifies this phenomenon. Notably, qubit $|q4\rangle$ is initialised initially but remains inactive for four subsequent operations. Consequently, the IdQ value for this circuit configuration is 4.

7. **Non-zero Initialization**: Qubits are initialized to non-zero states even when $|0\rangle$ would be sufficient.

   - *Identification Criteria*: Explicit initialization to $|1\rangle$ or superposition.
   - *Key Indicator of Severity*: Adds unnecessary operations and potential preparatory mismatches.

$$\text{NZI} \;=\; |\{\, q \in Q \mid \text{init}(q) \neq |0\rangle \}|$$

It is calculated by counting how many qubits are explicitly prepared into $|1\rangle$, superpositions, or other states; cardinality gives the total extent of non-default initialization. The most stable state of a qubit is typically observed as $|0\rangle$. However, during initialisation and subsequent gate operations, a qubit's state might transiently shift to an unstable state. Notably, this transition occurs when actions on other qubits separate the qubit's initialisation and gate operations. This scenario is termed the Initialization of Qubits (IQ). IQ is primarily different from IdQ as in the former, the difference between the first and the second operation matters, while in the latter, the maximum time a qubit is left alone is significant. The Figure 2.6 illustrates the execution matrix of the Bernstein Vazirani Circuit. This circuit efficiently discovers a secret binary number, outperforming classical computers. While a classical computer requires O(n) time to uncover the n-bit private number, this algorithm accomplishes the task in O(1) time. In the depicted figure, qubit $|q2\rangle$ is initialised at the circuit's commencement, and a Hadamard gate(x()) gate is applied to it. Yet, the

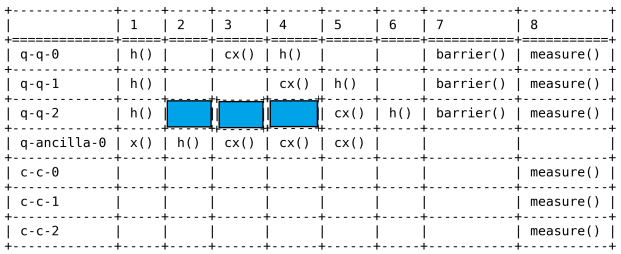


Figure 2.6: IQ- Bernstein Vazirani circuit

subsequent gate application to this qubit follows three operations. Consequently, this particular circuit exhibits an IQ value of 3, indicating a potential QCS.

8. **Misalignment Between Logical and Physical Qubits**: Poor mapping that introduces unnecessary routing overhead.

   - *Identification Criteria*: Excessive SWAP gates or degraded mapping to hardware topology.
   - *Key Indicator of Severity*: Heightened gate count and circuit depth.

$$\text{MAP_SWAPs} \;=\; |\{\, g \in G_{\text{routed}} \mid \text{type}(g) = \text{SWAP}\}|$$

$$\Delta\text{Depth} \;=\; \max_{\pi \in \mathcal{P}_{\text{routed}}} |\pi| \;-\; \max_{\pi \in \mathcal{P}_{\text{unrouted}}} |\pi|$$

   After mapping/logical routing, G_{routed} contains inserted SWAPs to satisfy hardware connectivity; counting them measures routing overhead. The depth increase ΔDepth is the difference between routed and unrouted longest-path lengths.

9. **Superfluous Operations (bug-pattern smell)**: Gates that have no impact on the final output distribution.

   - *Identification Criteria*: Removing the gate does not alter output probability distribution.
   - *Key Indicator of Severity*: Unnecessary noise with no functional contribution.

$$\text{SO} \;=\; \left| \left\{ g \in G \;\middle|\; \Pr(\mathbf{z}|C) = \Pr(\mathbf{z}|C \setminus \{g\}),\ \forall\, \mathbf{z} \right\} \right|$$

   Where $\Pr(\mathbf{z} \,|\, \cdot)$ is the probability of observing bitstring $\mathbf{z}$ on the measured qubits M. A gate is superfluous if removing it leaves the entire output distribution unchanged; counting such gates quantifies useless operations.

10. **Idle or Unused Qubits (bug-pattern smell)**: Qubits that are never used or never influence measured outcomes.

- *Identification Criteria*: No gates act on the qubit or operations don't affect measurements.
- *Key Indicator of Severity*: Wasted resources and inflated circuit size.

$$\mathrm{UUQ} \;=\; |\{\, q \in Q \mid \deg_G(q) = 0 \text{ or } q \not\rightsquigarrow M \}|$$

Where $\deg_G(q)$ is the number of gates touching q and $q \not\rightsquigarrow M$ indicates no dependency path to measurement.

Some of the remaining newly introduced quantum-specific code smells are described in Section 6.3.4.

2.5 Data Analysis Methods

This work utilized data analysis approaches from three key domains: ML, (DL), and evolutionary algorithms. ML approaches use historical data to find patterns and model relationships between independent and dependent variables, without using preconceived equations. DL, a specialized branch of ML, uses neural networks with numerous layers to automatically extract complicated characteristics and patterns from big datasets. It is particularly successful for complex tasks like picture recognition and natural language processing. Evolutionary algorithms are inspired by the natural selection process, and they use principles including selection, mutation, and crossover to iteratively refine solutions to complicated problems. The models created using these techniques are then utilized to forecast outcomes for future data points.

2.5.1 Machine Learning Classifiers

ML classifiers are fundamental tools for assigning data instances to predefined categories by utilizing patterns extracted from training datasets. Commonly employed techniques include Decision Tree (DT)s, SVM, and Random Forest (RF)s, each designed to handle specific types of learning problems [79]. In general, classifiers fall into two major groups: supervised and unsupervised.

Supervised learning algorithms, such as Logistic Regression (LR) and SVM, build predictive models using labeled datasets, where both inputs and their corresponding outputs are provided [80]. These models aim to generalize from training examples to deliver accurate predictions for unseen data. On the other hand, unsupervised methods, such as k-means clustering, function without labeled outputs and attempt to uncover natural groupings or structures within the data [81]. Such approaches are especially valuable in

exploratory data analysis, where hidden relationships must be identified without prior knowledge of class labels.

More recently, ensemble strategies have become increasingly influential in ML. By integrating the predictions of multiple base classifiers, ensemble methods enhance accuracy and robustness [82]. This collective approach reduces the weaknesses of individual learners, resulting in more dependable and precise outcomes.

2.5.1.1 Decision Tree

A DT is a supervised, non-parametric learning approach that recursively divides the input space into smaller subregions, each of which ideally contains samples belonging to the same class. Formally, consider a dataset

$$\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}, \quad x_i \in \mathbb{R}^d,\ y_i \in \mathcal{C}, \tag{2.1}$$

where x_i denotes the d-dimensional feature vector of the i-th observation, y_i is its associated class label, and $\mathcal{C}$ represents the set of all possible classes.

The internal nodes of the tree are created by applying split criteria that evaluate how well a feature separates the data. Widely adopted criteria include Entropy, Information Gain and Gini Index.

Entropy is a measure of impurity or uncertainty in a dataset, indicating how mixed the class labels are. It is defined as :

$$H(S) = -\sum_{c \in \mathcal{C}} p(c)\ \log_2 p(c), \tag{2.2}$$

where S is a subset of the training set, and $p(c)$ indicates the relative frequency of class c in S.

Information Gain describes the reduction in entropy achieved by splitting a dataset on a particular attribute, used to select the best feature in decision trees. It is represented as:

$$IG(S, A) = H(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|}\, H(S_v), \tag{2.3}$$

with A as the candidate feature, Values(A) representing the possible values of A, and S_v denoting the subset of S corresponding to $A = v$.

Gini Index is a measure of node impurity that quantifies the probability of incorrectly classifying a randomly chosen instance if it were labeled according to the class distribution of the node. It is calculated as:

$$Gini(S) = 1 - \sum_{c \in \mathcal{C}} \Big(p(c)\Big)^2, \tag{2.4}$$

where $p(c)$ is the observed probability of class c in S.

At each split, the feature A^* that achieves the highest discrimination is chosen:

$$A^* = \arg\max_{A \in \mathcal{F}} \; IG(S, A), \tag{2.5}$$

where $\mathcal{F}$ is the set of candidate features.

The recursive process continues until a termination criterion is reached (e.g., tree depth limit, minimum number of samples in a node, or perfectly pure subsets). The final decision at each leaf is determined by majority voting among training samples:

$$\hat{y}(x) = \arg\max_{c \in \mathcal{C}} \; p(c \,|\, x \in \text{leaf}), \tag{2.6}$$

where $\hat{y}(x)$ is the predicted class for input x, and $p(c \,|\, x \in \text{leaf})$ gives the relative frequency of class c within that leaf.

In this work, a conventional DT model is employed, where the tree expands by iteratively choosing the feature that provides the best split at each node. This mechanism enables the model to capture the underlying structure of the dataset while retaining interpretability. Unlike many complex algorithms, DTs are highly transparent since their decision logic can be represented in a straightforward tree format, which is easy to visualize and interpret. A notable strength of DTs is their ability to handle both numerical and categorical attributes, making them suitable for diverse application areas [83]. On the downside, deep trees often suffer from overfitting, though strategies such as pruning can alleviate this issue [84]. Despite these limitations, DTs remain widely used due to their simplicity, flexibility, and interpretability, particularly in domains where explainability of predictions is essential.

2.5.1.2 Random Forest

RF is an ensemble-based technique that aggregates the outputs of multiple DTs to enhance prediction accuracy and mitigate overfitting. The method constructs a collection of trees during the training phase and derives the final output either by majority voting in classification tasks or by averaging predictions in regression tasks. Each tree is trained on a randomly sampled subset of the original dataset, a process known as bootstrap aggregating (bagging), which lowers variance and improves generalization. Additionally, at each split within a tree, only a randomly chosen subset of features is evaluated, thereby increasing diversity among the trees and contributing to a more reliable overall model [5].

Consider a dataset

$$\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}, \quad x_i \in \mathbb{R}^d,\ y_i \in \mathcal{C}, \tag{2.7}$$

RF builds T decision trees $\{h_t(x)\}_{t=1}^{T}$, each trained on a bootstrap sample of $\mathcal{D}$ and a random subset of features at each split.

The final prediction is obtained by majority voting (classification) or averaging (regression):

$$\hat{y}(x) = \arg\max_{c \in \mathcal{C}} \sum_{t=1}^{T} \mathbb{I}\Big(h_t(x) = c\Big), \tag{2.8}$$

where $\mathbb{I}(\cdot)$ is the indicator function, $h_t(x)$ is the prediction of the t-th tree, and T is the total number of trees.

The ensemble nature of RFs allows them to leverage the strengths of multiple trees while mitigating the weaknesses of individual trees. This method excels in handling large datasets with numerous features, as it can manage both numerical and categorical data without the need for extensive preprocessing or transformation [85]. Additionally, RFs provide built-in mechanisms for assessing feature importance, making them particularly useful for identifying the most influential variables in a dataset. Despite their advantages, RF models can be less interpretable than individual DTs due to their complexity, but their predictive performance often outweighs this drawback, making them a popular choice in various applications ranging from finance to healthcare.

2.5.1.3 Support Vector Machines (SVM)

Support Vector Machines (SVM) are supervised learning techniques widely applied to both classification and regression problems. The core principle of an SVM is to determine an optimal hyperplane that separates different classes of data while maximizing the margin between the boundary and the nearest training points, known as support vectors. Maximizing this margin contributes to improved generalization and robust classification. Beyond linear separation, SVMs are capable of handling non-linear decision boundaries through the use of kernel functions, which project the input data into higher-dimensional spaces where separability becomes feasible. This kernel-based approach allows SVMs to effectively model complex relationships. Furthermore, the training process involves solving a convex optimization problem, ensuring that the global optimum solution is attained. SVMs are particularly advantageous in high-dimensional spaces, making them suitable for tasks involving large feature sets.

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i \quad \text{subject to} \quad y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 1 - \xi_i,\ \xi_i \geq 0 \tag{2.9}$$

Here, $\mathbf{w}$ is the weight vector defining the hyperplane, b is the bias, $y_i \in \{-1, +1\}$ are class labels, ξ_i are slack variables allowing soft-margin violations, and $C > 0$ is the penalty parameter balancing margin maximization and misclassification errors.

In this study, several SVM classifiers with different kernel functions are used. The **Linear SVM** employs a linear kernel and is appropriate for linearly separable data, often used in high-dimensional spaces such as text classification [86]. The **Polynomial SVM** applies a polynomial kernel, allowing the model to capture more complex relationships in the data by transforming it into a higher-degree polynomial feature space [87]. The **Sigmoid SVM** uses a sigmoid kernel, which behaves similarly to Artificial Neural Network (ANN)s and is effective for non-linear classification tasks [88]. The **Radial Basis Function (RBF) SVM**, one of the most widely used kernels, transforms data into an infinite-dimensional space, making it highly effective for handling complex, non-linear classification problems [89]. The use of different kernels in SVM allows for flexibility in addressing various classification challenges, and the regularization parameter C plays a critical role in balancing training accuracy and generalization.

2.5.1.4 K-Nearest Neighbour

The K-Nearest Neighbour (KNN) algorithm is a simple, non-parametric, and instance-based learning method that is widely applied in both classification and regression tasks. For a given query sample, the algorithm identifies the k nearest neighbors in the feature space according to a distance metric, such as Euclidean, Manhattan, or Minkowski distance. In classification, the predicted label is determined by the majority class among the selected neighbors, while in regression the output is typically the mean (or sometimes a weighted mean) of their values. The method is intuitive and interpretable since its predictions are directly based on the distribution of the training data [90]. Among the various distance measures, Euclidean distance is most frequently employed to compute similarity between points.

$$d(\mathbf{x}, \mathbf{x}_i) \ = \ \sqrt{\sum_{j=1}^{m}(x_j - x_{ij})^2} \tag{2.10}$$

where $d(\mathbf{x}, \mathbf{x}_i)$ is the distance between the test instance $\mathbf{x}$ and training sample $\mathbf{x}_i$, and m denotes the number of features. The predicted class $\hat{y}$ is determined as:

$$\hat{y} \;=\; \arg\max_{c} \sum_{i=1}^{k} \mathbf{1}(y_i = c) \tag{2.11}$$

Here, $\mathbf{1}(\cdot)$ is an indicator function that counts how many of the k neighbors belong to class c.

One of the advantages of KNN is its adaptability to different types of datasets and its effectiveness in cases where the decision boundary is non-linear. Additionally, KNN does not require any training phase, making it well-suited for applications with lower data volumes or dynamic environments where data constantly changes [91]. However, the algorithm can become computationally intensive as the dataset size grows, especially during the prediction phase, since it must calculate the distance from the query point to all instances in the dataset. Another limitation is its sensitivity to irrelevant or redundant features, which can adversely affect accuracy. To mitigate this, dimensionality reduction techniques such as Principal Component Analysis (PCA) can be applied before using KNN [92].

While KNN's simplicity and flexibility make it a popular choice, its performance is generally more effective with smaller datasets and in low-dimensional spaces. Nonetheless, its ease of implementation and effectiveness in various applications, such as image recognition and recommendation systems, contribute to its broad usage in ML [93].

2.5.1.5 MultiLayer Perceptron

A MultiLayer Perceptron (MLP) is a feedforward architecture within the family of ANNs, consisting of an input layer, one or more hidden layers, and an output layer. Neurons in an MLP are fully connected, meaning each neuron in one layer is linked to every neuron in the subsequent layer. The training of MLPs is typically carried out using backpropagation, where the weights are updated by propagating the gradient of the loss function from the output layer back through the network to minimize prediction error.

The forward propagation for layer l can be expressed as:

$$h^{(l)} = \sigma\Big(W^{(l)} h^{(l-1)} + b^{(l)}\Big), \tag{2.12}$$

where $h^{(l)}$ denotes the output of layer l, $W^{(l)}$ is the weight matrix, $b^{(l)}$ is the bias vector, and $\sigma(\cdot)$ is a non-linear activation function such as ReLU, sigmoid, or tanh. Model optimization is achieved by minimizing a loss function (e.g., cross-entropy for classification) through backpropagation combined with Stochastic Gradient Descent (SGD).

MLPs are effective at modeling complex, non-linear relationships in data since each hidden layer introduces an additional level of abstraction, enabling the network to cap-

ture intricate patterns. This property makes them suitable for diverse applications, including image recognition, speech processing, natural language understanding, and predictive modeling. Nonetheless, they are susceptible to overfitting in high-dimensional feature spaces, which can be alleviated using strategies such as dropout, weight decay, and early stopping [94].

A notable theoretical strength of MLPs is their universality: according to the Universal Approximation Theorem, they can approximate any continuous function given sufficient neurons and training data [95]. However, deeper networks may become computationally expensive due to the large number of parameters involved. Despite these challenges, MLPs remain widely adopted because of their flexibility, expressive power, and proven effectiveness across multiple domains [96].

2.5.1.6 Gradient Boosting

Gradient Boosting (GB) is an ensemble learning technique that builds predictive models through an iterative process of adding weak learners, typically DTs, in a sequential manner. At each stage, the algorithm aims to minimize the error by focusing on instances where previous models underperformed. This is achieved by fitting a new learner to the residuals (the difference between actual and predicted values) of the existing model, with the objective of reducing overall prediction error [97].

The model prediction after M iterations is:

$$F_M(x) = F_0(x) + \sum_{m=1}^{M} \gamma_m h_m(x), \tag{2.13}$$

where

- $F_0(x)$ is the initial prediction (e.g., mean for regression, log-odds for classification),
- $h_m(x)$ is the weak learner (usually a shallow decision tree) at iteration m,
- γ_m is the learning rate or step size for $h_m(x)$,
- M is the total number of boosting iterations.

At each stage m, the weak learner $h_m(x)$ is trained to approximate the negative gradient of the loss function $\mathcal{L}(y, F(x))$:

$$r_{im} = -\left[\frac{\partial \mathcal{L}(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}, \tag{2.14}$$

where r_{im} is the pseudo-residual for sample i at iteration m.

The final model prediction for classification is obtained as:

$$\hat{y}(x) = \arg\max_{c \in \mathcal{C}} \; P(y = c \,|\, x) = \arg\max_{c \in \mathcal{C}} \; \text{softmax}\Big(F_M(x)\Big). \tag{2.15}$$

GB combines weak learners into a strong classifier by optimizing the loss function iteratively, balancing bias and variance.

The effectiveness of GB lies in its ability to adaptively improve model accuracy by prioritizing difficult-to-predict cases. It excels at handling both regression and classification tasks, is robust to various types of data distributions, and performs well even with high-dimensional data. However, GB can be computationally intensive, especially when dealing with large datasets, due to the sequential nature of the learning process. Techniques like early stopping, learning rate tuning, and regularization are commonly applied to enhance efficiency and prevent overfitting [98].

GB has led to the development of several powerful extensions, such as Extreme Gradient Boosting (XGB) and LightGBM, which introduce optimizations like tree pruning, efficient memory usage, and parallel processing to increase speed and accuracy [99]. While the interpretability of GB models can be more challenging compared to simpler methods, tools such as feature importance metrics and SHAP values are often used to gain insights into which features most influence the model's predictions.

2.5.1.7 XGB Classifier

The Extreme Gradient Boosting (XGB) classifier is an ensemble learning method based on gradient-boosted decision trees. It builds trees sequentially, where each new tree corrects the residual errors of the previous ensemble. The prediction rule is:

$$\hat{y}_i = \arg\max_{k \in \{1,\ldots,K\}} \sum_{t=1}^{T} f_t(x_i), \quad f_t \in \mathcal{F}, \tag{2.16}$$

where T is the total number of trees, f_t is the decision tree at iteration t, and $\mathcal{F}$ is the space of regression trees. The model is trained by minimizing the following regularized objective:

$$\mathcal{L}(\phi) = \sum_i l\Big(y_i, \hat{y}_i\Big) + \sum_{t=1}^{T} \Omega(f_t), \quad \Omega(f) = \gamma T + \tfrac{1}{2}\lambda \|w\|^2, \tag{2.17}$$

where l is a differentiable loss function, $\Omega(f)$ is the regularization term, γ penalizes the number of leaves, and λ controls the complexity of leaf weights.

XGBoost is particularly useful for code smell detection because it can handle large and imbalanced datasets while capturing subtle relationships between software metrics. Its ensemble of boosted decision trees is capable of modeling the non-linear and com-

plex patterns often present in smelly versus clean code. By providing high accuracy and robustness against overfitting, XGB ensures reliable detection of both common and rare smells. In addition, its built-in feature importance scores help researchers and practitioners understand which code attributes contribute most to the prediction, making the results more interpretable.

2.5.1.8 Naive Bayes

Naive Bayes belongs to a family of probabilistic classifiers that are grounded in Bayes' theorem and rely on the simplifying assumption that features are conditionally independent given the class label. This assumption, though often unrealistic, significantly reduces computational complexity and makes the method efficient for large-scale problems, particularly in text classification. The fundamental expression for Bayes' theorem is:

$$P(C \mid X) = \frac{P(X \mid C)\, P(C)}{P(X)} \tag{2.18}$$

where $P(C \mid X)$ is the posterior probability of class C given the feature vector X, $P(X \mid C)$ is the likelihood of observing X under class C, $P(C)$ is the prior probability of C, and $P(X)$ is the evidence. Since $P(X)$ is constant across classes, Naive Bayes focuses on maximizing the numerator under the independence assumption.

In this study, three popular variants of Naive Bayes are considered:

Bernoulli Naive Bayes (BNB): Designed for binary or boolean features, this model is effective in text applications where features represent the presence or absence of a word in a document [100].

Gaussian Naive Bayes (GNB): Assumes that the distribution of features within each class follows a Gaussian distribution. The likelihood of a feature X_i given class C is computed as:

$$P(X_i \mid C) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(X_i - \mu)^2}{2\sigma^2}\right), \tag{2.19}$$

where X_i is the ith feature, μ and σ^2 denote the class-specific mean and variance, respectively [101].

Multinomial Naive Bayes (MNB): Particularly suitable for text classification, this variant treats features as word counts or term frequencies. The likelihood of a document under class C is given by:

$$P(X \mid C) \;=\; \frac{(n+K)!}{n_1!\, n_2! \cdots n_K!} \prod_{k=1}^{K} P(w_k \mid C)^{\,n_k}, \tag{2.20}$$

where $n = \sum_{k=1}^{K} n_k$ is the total number of words in the document, K is the vocabulary size, n_k is the frequency of the kth word w_k, and $P(w_k \mid C)$ is its conditional probability under class C [102].

The prediction rule for Naive Bayes is obtained by selecting the class with the maximum posterior probability:

$$\hat{y} = \arg \max_{k \in \{1,\ldots,K\}} P(C_k) \prod_{j=1}^{d} P(x_j \mid C_k), \tag{2.21}$$

where $P(C_k)$ is the prior probability of class C_k, x_j is the jth feature, and $P(x_j \mid C_k)$ is the likelihood of x_j given C_k.

Despite its reliance on strong independence assumptions, Naive Bayes remains an efficient, interpretable, and robust classifier, especially effective in high-dimensional problems such as document categorization.

2.5.1.9 Logistic Regression (LR)

LR is a statistical model used primarily for binary classification tasks, where the objective is to predict the probability of a categorical outcome, typically represented as either 0 or 1. Unlike linear regression, which predicts continuous values, Logistic Regression employs the logistic (sigmoid) function to map predictions into the probability range $[0, 1]$. This probabilistic interpretation makes Logistic Regression particularly suitable for binary classification problems [103].

For binary classification ($y \in \{0, 1\}$), the probability of class 1 is given by the sigmoid function:

$$P(y = 1 \mid x) = \sigma(z) = \frac{1}{1 + e^{-z}}, \tag{2.22}$$

where

$$z = w^T x + b, \tag{2.23}$$

$w \in \mathbb{R}^d$ is the weight vector, b is the bias term, and $x \in \mathbb{R}^d$ is the feature vector.

The predicted class label is determined as:

$$\hat{y}(x) = \begin{cases} 1 & \text{if } P(y = 1 \mid x) \geq 0.5, \\ 0 & \text{otherwise.} \end{cases} \tag{2.24}$$

Model parameters are learned by minimizing the negative log-likelihood (cross-entropy

loss):

$$\mathcal{L}(w, b) = -\sum_{i=1}^{n} \Big[y_i \log P(y_i = 1 \mid x_i) + (1 - y_i) \log \Big(1 - P(y_i = 1 \mid x_i) \Big) \Big], \tag{2.25}$$

where $y_i \in \{0, 1\}$ denotes the true class label of sample x_i.

Logistic Regression works by modeling a linear relationship between input features and the log-odds of the target class. The sigmoid function then transforms these log-odds into probabilities, enabling classification. The model is computationally efficient, interpretable, and robust, with coefficients indicating the strength and direction of feature influence on the prediction [104].

Despite its simplicity, LR is effective for linearly separable data and can be enhanced with regularization techniques such as L1 or L2 regularization to prevent overfitting. However, it struggles with non-linear relationships. In such cases, extensions like polynomial features or kernel methods are employed to capture complex decision boundaries [105].

2.5.1.10 Stochastic Gradient Descent

SGD is an optimization algorithm commonly used for large-scale ML tasks due to its computational efficiency. Unlike standard Gradient Descent, which calculates the gradient over the entire dataset. It optimizes a loss function iteratively using stochastic updates. The parameter update rule is:

$$w^{(t+1)} = w^{(t)} - \eta_t \nabla \ell \Big(w^{(t)}; x_i, y_i \Big), \tag{2.26}$$

where $w^{(t)}$ is the weight vector at iteration t, η_t is the learning rate, and $\nabla \ell(\cdot)$ is the gradient of the loss function (e.g., hinge loss, log loss) with respect to a training sample (x_i, y_i). This allows efficient optimization for large-scale datasets. [106].

SGD's stochasticity allows it to escape shallow local minima but can cause convergence instability. Techniques such as momentum and adaptive learning rates (e.g., Adam) are used to improve performance [107].

2.5.1.11 Linear Discriminant Analysis

Linear Discriminant Analysis (LDA) is a supervised dimensionality reduction technique often used for classification tasks. LDA projects high-dimensional data onto a lower-dimensional space by maximizing the separation between different classes. It projects data onto a lower-dimensional space while maximizing class separability. The discrimi-

nant function for class k is:

$$\delta_k(x) = x^\top \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k^\top \Sigma^{-1} \mu_k + \ln \pi_k, \tag{2.27}$$

where x is the feature vector, μ_k is the mean of class k, Σ is the common covariance matrix across classes, and π_k is the prior probability of class k. A new observation is assigned to the class with the maximum discriminant score. This technique not only reduces computational complexity but also enhances class separability, making it effective in tasks with a limited number of features relative to samples [108].

2.5.1.12 ExtraTrees

The Extremely Randomized Trees classifier is an ensemble method similar to RFs but introduces more randomness in feature selection and split thresholds. Unlike traditional decision trees, which choose the best split based on a criterion like Gini impurity or entropy, ExtraTrees introduces randomness by selecting splits at random points within feature space. This method helps to reduce variance and mitigate overfitting, especially when working with noisy data. Given N random splits per feature and multiple trees in the ensemble, ExtraTrees effectively captures diverse patterns within the data and performs well on high-dimensional datasets [109]. The decision function of the ensemble is:

$$\hat{y} = \arg \max_{k \in \{1,\ldots,K\}} \frac{1}{T} \sum_{t=1}^{T} \mathbf{1}\Big(f_t(x) = k\Big), \tag{2.28}$$

where T is the total number of trees, $f_t(x)$ is the prediction of the t-th tree for input x, and $\mathbf{1}(\cdot)$ is the indicator function. Each tree is constructed using a randomly chosen subset of features and randomly chosen split thresholds, which helps in variance reduction and improves generalization.

2.5.2 Deep Learning Classifiers

DL classifiers have revolutionized data categorization by leveraging complex neural network architectures to learn hierarchical representations from raw data. Unlike traditional ML classifiers, DL models can automatically learn features, which makes them particularly effective for large-scale and high-dimensional data such as images, audio, and text [110]. Common DL classifiers include CNNs, RNNs, and Transformers, each designed for specific types of tasks and data structures [96].

Supervised DL models, such as CNNs for image classification and RNNs for sequential data, require large labeled datasets for training. By learning complex patterns in the

data, these models can achieve state-of-the-art performance in tasks like image recognition, natural language processing, and speech recognition [111]. In contrast, unsupervised and self-supervised DL techniques, such as autoencoders and generative adversarial networks (GANs), do not require labeled data. These methods excel at discovering latent structures in the data and generating realistic samples, making them valuable for tasks like anomaly detection and data synthesis [112].

Recently, ensemble techniques have also been applied to DL to further enhance model robustness and generalization. By combining multiple neural networks, ensemble methods help reduce variance and mitigate overfitting, leading to more reliable predictions [113]. In this study, we utilize DL classifiers to analyze intricate data structures, uncover essential patterns, and boost predictive accuracy. These advanced techniques enable us to derive deeper insights and improve decision-making processes.

2.5.2.1 Convolutional Neural Network

Convolutional Neural Networks (CNNs) are a specialized type of neural network designed to process and analyze grid-like data structures, particularly images. CNNs leverage convolutional layers to automatically extract spatial and hierarchical features, making them highly effective for tasks such as image classification, object detection, and semantic segmentation [114]. The core operation in a Convolutional Neural Network (CNN) is the convolution, which can be mathematically expressed as:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) \cdot K(m,n) \tag{2.29}$$

where

- $S(i,j)$ represents the output feature map at position (i,j).
- I denotes the input image or input feature map.
- K corresponds to the kernel (or filter), which is a smaller learnable matrix applied across the input.
- (i,j) denotes the spatial location in the output feature map.
- m,n represent the indices of the kernel dimensions.

The convolution operation involves sliding the kernel K across the input I and performing element-wise multiplications followed by summation. This enables CNNs to capture local spatial features such as edges, corners, and textures, which can be hierarchically combined in deeper layers to detect complex patterns. In the context of textual data,

the convolution operation captures local patterns in sequences (such as n-grams) by applying filters over word embeddings or character-level representations, thereby enabling the model to learn context-specific features useful for classification or sequence modeling tasks.

A CNN architecture typically includes multiple convolutional, pooling, and fully connected layers, each playing a role in learning increasingly complex patterns within the data [111]. CNNs capture local dependencies, allowing them to learn translationally invariant features, making them robust to variations in position, scale, and orientation of objects within images [115]. Advanced CNN architectures, such as ResNet and Inception, further enhance performance through techniques like skip connections and multi-scale feature extraction, enabling deeper networks and improving accuracy on complex datasets [113].

CNNs are highly effective for text classification tasks as they can automatically capture local features such as n-grams without the need for explicit manual feature engineering. By applying convolutional filters over word embeddings, CNNs are able to detect semantic patterns like key phrases or contextual cues, which are crucial for textual understanding. Moreover, CNNs are computationally efficient due to their weight-sharing mechanism and the ability to process text in parallel, making them particularly suitable for large-scale textual datasets. Their hierarchical feature extraction also enables them to learn increasingly abstract semantic representations as the network deepens, which enhances the overall performance on tasks such as sentiment analysis, topic categorization, and document classification.

2.5.2.2 Long Short Term Memory

Long Short-Term Memory (LSTM) networks are a specialized type of RNN designed to capture long-range dependencies in sequential data by employing a system of gates [116]. Unlike traditional RNNs, which often struggle with the vanishing gradient problem, LSTMs introduce dedicated mechanisms to regulate the flow of information through time [117]. Each LSTM cell includes an input gate, a forget gate, and an output gate, which collectively determine how information is stored, updated, and exposed to the next time step.

The operations within an LSTM cell are given by:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f), \quad i_t = \sigma(W_i[h_{t-1}, x_t] + b_i), \tag{2.30}$$

$$\tilde{c}_t = \tanh(W_c[h_{t-1}, x_t] + b_c), \quad c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t, \tag{2.31}$$

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o), \quad h_t = o_t \cdot \tanh(c_t), \tag{2.32}$$

where:

- f_t (**forget gate**) decides which parts of the previous memory state c_{t-1} should be preserved.
- i_t (**input gate**) controls how much new information from the candidate state $\tilde{c}_t$ is incorporated.
- c_t is the **cell state**, functioning as the memory line that carries relevant information through time.
- o_t (**output gate**) determines how much of the memory is exposed as the hidden representation.
- h_t is the **hidden state**, which is used to generate predictions at each step.

By selectively remembering and forgetting, LSTMs overcome the limitations of standard RNNs and are capable of modeling long-term dependencies. Their ability to capture contextual relationships makes them highly effective in sequential tasks such as language modeling, sentiment classification, and machine translation.

LSTMs offer significant advantages in textual data processing because they can effectively capture long-range dependencies across sequences of words, which is essential in natural language tasks where the meaning of a sentence often depends on distant contextual words. Unlike traditional RNNs, LSTMs overcome the vanishing gradient problem through the gating mechanism, ensuring that relevant information is preserved over long sequences. This makes them highly effective for tasks such as language modeling, text generation, and machine translation, where contextual information across multiple time steps is critical. Furthermore, LSTMs can adaptively decide what information to retain or discard, which improves their ability to model complex linguistic structures inherent in natural language.

2.5.2.3 Gated Recurrent Unit

Gated Recurrent Units (GRUs) are a type of recurrent neural network similar to LSTMs but with a simpler structure. GRUs use two gates an update gate and a reset gate to manage information flow, making them computationally efficient while still effectively capturing dependencies in sequential data [118]. GRUs are particularly effective in tasks

involving sequential patterns, such as language processing and time-series forecasting [6].

The core operations within a GRU cell are given by:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z) \tag{2.33}$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r) \tag{2.34}$$

$$\tilde{h}_t = \tanh(W_h \cdot [r_t \cdot h_{t-1}, x_t] + b_h) \tag{2.35}$$

$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t \tag{2.36}$$

where:

- z_t is the **update gate**, which controls how much of the previous hidden state h_{t-1} should be carried forward to the current state.
- r_t is the **reset gate**, which decides how much of the past information h_{t-1} should be ignored when computing the candidate hidden state $\tilde{h}_t$.
- $\tilde{h}_t$ is the **candidate hidden state**, combining the current input x_t with the selectively reset previous hidden state.
- h_t is the **final hidden state**, obtained by blending h_{t-1} and $\tilde{h}_t$ based on the update gate z_t.

This gating mechanism enables GRUs to capture long-term dependencies with fewer parameters compared to LSTMs, making them computationally efficient for sequence modeling tasks such as natural language processing and speech recognition.

GRUs are advantageous in textual data processing because they retain the ability to model long-term dependencies while being more computationally efficient than LSTMs due to their simplified architecture. The update and reset gates allow GRUs to control the flow of information without requiring a separate memory cell, making them faster to train while maintaining comparable performance to LSTMs on most natural language tasks. Their reduced number of parameters also helps mitigate the risk of overfitting, especially in scenarios with limited training data. GRUs are particularly effective in tasks such as text classification, sequence labeling, and question answering, where they provide a balance between computational efficiency and the ability to capture semantic dependencies in textual sequences.

2.5.2.4 Bi-LSTM

Bidirectional Long Short-Term Memory (Bi-LSTM) networks are an extension of the standard LSTM architecture designed to capture both past and future context in sequential data. Unlike a unidirectional LSTM that processes input in a single direction (forward), a Bi-LSTM has two parallel layers: one that processes the sequence forward and another that processes it backward. The outputs from both directions are concatenated to form the final representation.

The forward and backward hidden states are given as:

$$\overrightarrow{h_t} = LSTM(x_t, \overrightarrow{h_{t-1}}) \tag{2.37}$$

$$\overleftarrow{h_t} = LSTM(x_t, \overleftarrow{h_{t+1}}) \tag{2.38}$$

The final hidden state combines both directions:

$$h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}] \tag{2.39}$$

Bi-LSTM offers a significant advantage in text processing because natural language often depends on both preceding and succeeding words for proper interpretation. By incorporating context from both directions, Bi-LSTMs improve tasks such as sentiment analysis, machine translation, and named entity recognition.

2.5.2.5 GNN

GNNs are DL architectures specifically designed to learn from graph-structured data. Unlike sequential or grid-based models, GNNs can effectively represent entities as nodes and their relationships as edges, thereby capturing both local and global structures of the graph. In each layer of a GNN, node representations are iteratively updated by aggregating features from their neighbors.

$$h_v^{(k)} = \sigma\left(W^{(k)} \cdot \text{AGGREGATE}\left(\{h_u^{(k-1)} : u \in \mathcal{N}(v)\} \cup \{h_v^{(k-1)}\}\right)\right) \tag{2.40}$$

Here, $h_v^{(k)}$ denotes the representation of node v at the kth layer, $\mathcal{N}(v)$ represents the set of neighboring nodes of v, AGGREGATE$(\cdot)$ is a neighborhood aggregation function such as mean, sum, or max, $W^{(k)}$ are the trainable parameters of the layer, and σ is a nonlinear activation function.

GNNs are highly effective for textual data processing because text can be represented as graphs using syntactic dependency trees, co-occurrence graphs, or semantic links between words. This graph-based representation allows GNNs to capture contextual de-

pendencies beyond sequential word order, making them useful for tasks such as document classification, semantic role labeling, and relation extraction. They excel in modeling long-range dependencies and capturing relationships across different parts of a text.

2.5.2.6 GCN

Graph Convolutional Networks (GCNs) are a specialized form of GNNs that generalize the concept of convolution from Euclidean domains (such as images) to graph-structured data. Instead of sliding filters, GCNs use spectral graph theory to perform convolution operations by propagating and aggregating information across graph nodes.

The forward propagation rule of a GCN layer is expressed as:

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right) \tag{2.41}$$

where $H^{(l)}$ is the matrix of node features at layer l, $\tilde{A} = A + I$ is the adjacency matrix with added self-loops, $\tilde{D}$ is the diagonal degree matrix of $\tilde{A}$, $W^{(l)}$ represents the trainable weight matrix at the lth layer, and σ denotes an activation function such as ReLU.

GCNs are particularly advantageous in natural language processing tasks where graphs can be constructed from textual data. For example, word co-occurrence graphs, dependency parse trees, and semantic relation graphs benefit from GCNs' ability to propagate contextual information across connected nodes. Unlike traditional sequential models, GCNs can capture both local and global dependencies, making them powerful for tasks like relation extraction, text classification, and knowledge graph reasoning. Their ability to model complex structures makes them well-suited for representing textual data beyond linear word sequences.

2.5.2.7 GAT

Graph Attention Networks (GATs) enhance GCNs by introducing attention mechanisms during neighborhood aggregation. Instead of equally weighting all neighbors, GATs assign learnable attention coefficients that determine the relative importance of each neighboring node.

The attention mechanism is defined as:

$$e_{ij} = \text{LeakyReLU}(a^T[Wh_i \,\|\, Wh_j]) \tag{2.42}$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k\in\mathcal{N}(i)}\exp(e_{ik})} \tag{2.43}$$

$$h_i' = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} W h_j\right) \tag{2.44}$$

By adaptively focusing on the most relevant neighbors, GATs improve over GCNs in capturing nuanced contextual dependencies. In natural language processing, GATs excel in tasks such as semantic role labeling, relation extraction, and graph-based sentence representation, since attention highlights critical word or entity interactions that standard GCNs may underemphasize.

2.5.2.8 Ensemble Methods: Homogeneous and Heterogeneous

Ensemble methods combine multiple base models to improve predictive performance by reducing variance, bias, or improving generalization [3]. There are two main types of ensemble methods: homogeneous and heterogeneous.

In **homogeneous ensembles**, multiple instances of the same algorithm are used, and diversity is introduced through techniques like bagging, boosting, or stacking [4]. For example, RF is a homogeneous ensemble that combines multiple decision trees through bagging, averaging their predictions to reduce variance [5].

In contrast, **heterogeneous ensembles** combine different types of algorithms to capture a wider range of patterns within the data [1]. Techniques such as stacking often employ heterogeneous ensembles, where models like decision trees, SVMs, and neural networks are combined, and a meta-learner aggregates their outputs for the final prediction. Heterogeneous ensembles are useful when the individual models have complementary strengths, leading to improved accuracy and robustness.

By leveraging ensemble methods, this study aims to improve model stability and achieve a balance between bias and variance, enhancing the overall prediction performance.

2.5.2.9 BERT

BERT is a pre-trained deep bidirectional Transformer model designed to capture contextual relationships between words in text by jointly conditioning on both left and right contexts. It leverages the Transformer encoder architecture, where each layer applies multi-head self-attention and feed-forward operations to learn semantic representations.

For a given sequence of tokens $\{t_1, t_2, \ldots, t_n\}$, BERT encodes them into contextual embeddings:

$$H = \text{TransformerEncoder}(E(t_1), E(t_2), \ldots, E(t_n)), \tag{2.45}$$

where $E(t_i)$ is the input embedding of token t_i, and H denotes the sequence of hidden

states enriched with bidirectional context.

In the context of quantum code smell detection, BERT is employed through transfer learning. The pre-trained model, originally trained on large natural language corpora, is fine-tuned on quantum program datasets to learn structural and semantic patterns. This enables it to generalize effectively, providing a strong baseline for comparison against specialized DL architectures such as CNN, LSTM, and GRU.

2.5.3 Evolutionary Algorithms

Evolutionary Algorithms (EAs) represent a family of optimization methods inspired by biological evolution, where mechanisms such as selection, mutation, recombination, and survival of the fittest guide the search process [8]. These algorithms are particularly well-suited for tackling problems with large, complex, or poorly defined search spaces. By maintaining and evolving a population of candidate solutions, EAs are capable of approximating global optima through iterative refinement across generations.

The generic workflow of an EA can be summarized as follows:

1. **Initialization**: Create an initial set of candidate solutions, typically generated at random.
2. **Evaluation**: Measure the quality of each candidate using a fitness function $f(x)$.
3. **Selection**: Choose promising candidates, based on their fitness, to act as parents for producing the next generation.
4. **Variation**: Apply evolutionary operators, such as crossover (recombination) and mutation, to generate new offspring.
5. **Replacement**: Form the next population by combining or substituting existing individuals with newly generated offspring.

The process repeats until a termination criterion, such as a maximum number of generations or desired fitness level, is met.

2.5.3.1 Genetic Algorithm (GA)

The Genetic Algorithm (GA) is a well-known evolutionary algorithm that mimics the process of natural selection. In GA, each candidate solution is represented by a chromosome, typically a binary or real-valued vector. Genetic operators, such as crossover and mutation, are applied to evolve the population over generations [7].

The steps involved in GA can be expressed as follows:

$$
\begin{aligned}
x_{\text{new}} &= \text{crossover}(x_{\text{parent1}}, x_{\text{parent2}}) \\
x_{\text{mutated}} &= \text{mutation}(x_{\text{new}})
\end{aligned}
\tag{2.46}
$$

Here, x_{parent1} and x_{parent2} represent two parent chromosomes selected for reproduction. The crossover operation combines parts of the parents to create a new offspring x_{new}, and mutation introduces small, random alterations to produce x_{mutated}, which promotes diversity in the population. GA is widely used in fields such as engineering optimization, feature selection, and scheduling due to its robustness and adaptability.

In this study, a GA begins by generating binary vectors representing potential feature subsets. These subsets' fitness is evaluated using a customized function based on an ML classifier's performance. Individuals are selected for reproduction via Roulette Wheel Selection, mirroring natural selection. Single Point Crossover allows genetic recombination of features between parents, promoting diversity. Random Resetting Mutation further diversifies the feature space, preventing premature convergence. Evolved populations' fitness is reassessed, with superior individuals replacing less fit ones. This iterative process continues until convergence criteria are met, refining feature subsets and optimizing ML classifier accuracy through genetic operations.

2.5.3.2 Modified GA

The modified GA code incorporates specific modifications tailored to code smell detection given by Hemanth and Anita [119]. This approach adopts the best-performing variant proposed in their study, which combines the standard GA with a local search refinement mechanism, effectively enhancing the algorithm's capacity to balance exploration and exploitation. The process begins with the initialization of a population of encoded feature subsets. *Roulette Wheel Selection* is used to prioritize individuals based on their fitness scores, ensuring that fitter solutions have a higher probability of being selected for reproduction. A *Two-Point Crossover* operation is employed to generate new offspring, encouraging the exchange of genetic material between parent solutions and diversifying the search space. To maintain diversity and avoid premature convergence, *Flip Mutation* is applied to randomly toggle feature bits within the chromosome. Following these operations, a local search heuristic is executed on the best-performing individuals to refine their feature combinations, further improving classification accuracy. This modified GA approach iterates until convergence, resulting in more accurate and efficient feature selection for detecting code smells.

2.5.3.3 Particle Swarm Optimization

Particle Swarm Optimization Particle Swarm Optimization (PSO) given by Eberhart and Kennedy [120] mimics the social behaviour of birds and fish to optimize solutions. In PSO, particles representing potential solutions explore the search space collectively. Each particle adjusts its position based on its own and its neighbours' experiences, aiming for convergence to the optimal solution.

The PSO function used in the study begins by initializing a swarm of particles, where each particle represents a potential solution in the feature space for classification. Particle positions are randomly set within predefined bounds, and their velocities are initialized randomly. Equations are developed to update the particle position and velocity.

Update Velocity

$$\begin{aligned} \text{vel}_{ij}^{t+1} = & \, w \cdot \text{vel}_{ij}^{t} \\ & + c_1 \cdot r_1 \cdot (\text{Best_pos}_j - \text{pos}_{ij}^{t}) \\ & + c_2 \cdot r_2 \cdot (\text{Best_pos}_j - \text{pos}_{ij}^{t}) \end{aligned} \tag{2.47}$$

Update Position

$$\text{pos}_{ij}^{t+1} = \text{pos}_{ij}^{t} + \text{vel}_{ij}^{t+1} \tag{2.48}$$

where:

- pos_{ij}^{t} is the current position of particle i in dimension j at time t.
- pos_{ij}^{t+1} and vel_{ij}^{t+1} are respectively the updated position and velocity of particle i in dimension j at time $t+1$.

The PSO algorithm iteratively refines solutions over a set number of iterations. Particles update their velocities and positions based on current velocities, distances to personal best solutions, and distances to the global best solution within the swarm. This process encourages effective exploration of the solution space. Particle positions are constrained within specified bounds to prevent exceeding predefined limits. The fitness of particles in new positions is re-evaluated using a custom fitness function. The best global and individual solutions are updated based on the highest fitness values achieved. Convergence is monitored by recording the best fitness score at each iteration.

2.5.3.4 QIEA

The Quantum-Inspired Evolutionary Algorithm (QIEA) [2] presents a groundbreaking approach to optimization problems by drawing inspiration from the principles of QC. It

employs a novel ***qubit*** representation, utilizing quantum bits *(qubits)* that can exist in multiple states simultaneously due to superposition. A qubit is a composite of the smallest units of information called *q-bits*, forming a string expressed by Eq. 2.49.

$$q = (q_1|q_2|\dots|q_p|\dots|q_m) \tag{2.49}$$

Here, $p = \{1, 2, \dots, m\}$ and m corresponds to the number of q-bits. A q-bit (q_p) serves as a probabilistic representation, denoting the fundamental unit of information. In contrast to classical computer bits, which store either '0' or '1' at a given time, a q-bit can concurrently represent both '0' and '1' also by leveraging the concept of probability. Typically, a q-bit (q_p) is a probabilistically linear superposition of '0' and '1', as defined in Eq. 2.50.

$$q_p = \alpha_p|0\rangle + \beta_p|1\rangle \tag{2.50}$$

Where the parameters α_p and β_p represent two complex numbers denoting the probabilities of a q-bit being in the states "0" and "1," respectively. Consequently, adhering to the probability model, $|\alpha^2|$ and $|\beta^2|$ govern the likelihood of a q-bit residing in state "0" and state "1," normalizing to unity subject to the constraint mentioned in Eq. 2.51.

$$|\alpha_p^2| + |\beta_p^2| = 1; \quad 0 \leq \alpha_p \leq 1, \quad 0 \leq \beta_p \leq 1 \tag{2.51}$$

In the case of a single quantum bit (q) represented by a single q-bit (q_p) where $p = 1$, the system can exhibit two states, "0" and "1." Extending this to a quantum bit (q) with two q-bits (q_p), denoted as $p = \{1, 2\}$ in Eq. 2.52, allows for the representation of the linear superposition of four states ("00", "01", "10", and "11") simultaneously, as expressed in Eq. 2.53.

$$q = \langle {}^{\alpha_1}_{\beta_1}|^{\alpha_2}_{\beta_2}\rangle \tag{2.52}$$

$$\begin{aligned} q =& (\alpha_1 \times \alpha_2)\langle 00\rangle + (\alpha_1 \times \beta_2)\langle 01\rangle \\ &+ (\alpha_2 \times \beta_1)\langle 10\rangle + (\beta_1 \times \beta_2)\langle 11\rangle \end{aligned} \tag{2.53}$$

Similarly, a quantum bit (q) with m q-bits can represent 2^m states simultaneously. This fundamental attribute of QC endows the QIEA with distinct advantages over other metaheuristic algorithms. This includes enhanced population diversity, a reduced number of individuals required to navigate extensive search spaces, and a more efficient attainment of the global optimal solution. Additionally, QIEA observes and transforms quantum bits into classical individuals, and the comprehensive flow of the QIEA process is elucidated

in Algorithm 1. The algorithm alternates between two key processes: observation and updating. During the observation phase, the q-bit representation is measured, or collapsed to generate classical binary solutions based on the probability amplitudes. In the updating phase, these q-bits are adjusted using a rotation gate operator, which alters their probability amplitudes depending on the relative quality of the observed solution and the global best. The direction and angle of rotation are determined to bias the population toward better solutions, effectively guiding the search process.

Algorithm 1 Quantum Inspired Evolutionary Algorithm (QIEA)

Initialize: $q_N^{(t)}$
Generate $P_N^{(t)}$ by observing the quantum states in $q_N^{(t)}$
Evaluate the solutions in $P_N^{(t)}$
Preserve the best solutions from $P_N^{(t)}$ in $B_N^{(t)}$
while not meeting termination condition **do**
    $t \leftarrow t + 1$
    Generate $P_N^{(t)}$ by observing the quantum states in $q_N^{(t-1)}$
    Evaluate the solutions in $P_N^{(t)}$
    Update the quantum state $q_N^{(t)}$ using quantum gates
    Preserve the best solutions from $B_N^{(t-1)}$ and $P_N^{(t)}$ in $B_N^{(t)}$
    if meeting migration condition **then**
        Perform global or local migration
    end if
end while

In the QIEA, a population of N quantum bits ($q = \{q_1, q_2, \ldots, q_i, \ldots, q_N\}$) is initialized, where each q_i consists of n q-bits and defined as $[\alpha\,\beta]^T$ in terms of its quantum state. Further, it incorporates an *observation process*, similar to the measurement process in QC, where the quantum state is collapsed to obtain a definite classical state. During the tth iteration, the generated quantum bits (q_N^t) are used to obtain an equal number of binary representations (P_N^t) according to the observation process, as depicted in the Algorithm 2, while the fitness of every individual in the population is assessed using the fitness function f (which is specific to the problem in hand).

Within each iteration (t), quantum gate operations (***Q-gate***, acting as variation operators) are applied to update the states of the Q_N^t for each individual q_N using Eq. 2.54.

$$\alpha_i^{(t)} = [\cos(\Delta\theta) \cdot (\alpha_i^{(t-1)}) - (\sin(\Delta\theta) \cdot \beta_i^{(t-1)})] \tag{2.54}$$

Where $\Delta\theta$ is the rotation angle of each qubit toward either 0 or 1 state depending on its sign, and β value is determined as per Eq. 2.51. This quantum intervention, involving measurements and subsequent fitness evaluations, facilitates exploration and exploita-

Algorithm 2 Quantum Bit Observation Process

Input: Quantum bit q_i represented as $\{|q_1|q_2|\ldots|q_p|\ldots|q_m|\}$, where $q_p = [\alpha_p\, \beta_p]^T$.
Output: $X_i = \{x_1, x_2, \ldots, x_p, \ldots, x_m\}$: Binary representation of q_i.
while $p \leq m$ **do**
    if random[0,1) $\leq |\alpha_p|$ **then**
        Set $x_p = 1$
    else
        Set $x_p = 0$
    end if
end while

tion in the solution space. Finally, the update to B_N^t occurs through the migration process upon satisfying the migration condition. The migration condition serves as a design parameter capable of introducing variations in the quantum bits. QIEA incorporates two migration types: local migration and global migration. During local migration, some solutions within B_N^t are substituted with the best solution among them. In contrast, global migration involves replacing every solution in B_N^t with the global best solution from the tth iteration.

2.5.4 AutoML

Automated Machine Learning (AutoML) is a rapidly advancing field focused on automating the end-to-end process of applying machine learning to real-world problems [12]. AutoML tools aim to reduce the need for extensive human intervention by automatically selecting the best algorithms, optimizing hyperparameters, and performing feature engineering. This approach democratizes ML, enabling non-experts to build competitive models and experts to streamline their workflows. In this study, we explore three popular AutoML frameworks: TPOT, PyCaret, and H2O, each offering unique functionalities and strengths.

2.5.4.1 Tree-based Pipeline Optimization Tool (TPOT)

Tree-based Pipeline Optimization Tool (TPOT) is a Python-based AutoML library that uses genetic programming to automate the design of ML pipelines [9]. TPOT constructs complex pipelines by combining preprocessing, feature selection, and model selection, iterating through generations of solutions to optimize performance. Each pipeline configuration is evaluated based on its performance on a specified metric, with successful configurations being retained and recombined in subsequent generations.

The pipeline optimization process in TPOT can be expressed as:

$$\text{maximize } f(\theta) = \text{accuracy or other metric}$$

where θ represents the combination of preprocessing steps, feature selection methods, and model parameters. TPOT uses genetic operations like crossover and mutation to evolve these configurations across generations. This evolutionary approach enables TPOT to discover high-performing pipelines that may be non-intuitive or otherwise challenging for manual design.

2.5.4.2 PyCaret

PyCaret is an open-source, low-code AutoML library in Python designed for rapid prototyping and deployment of ML models [10]. PyCaret simplifies the machine learning workflow by providing an interface where users can set up an end-to-end pipeline with minimal code. It supports tasks like classification, regression, clustering, and anomaly detection, allowing users to evaluate multiple models, compare them, and deploy the best-performing ones.

The general process for setting up a model in PyCaret involves:

1. **Setup**: Initializes the dataset and predefines preprocessing steps.
2. **Model Training**: Evaluates multiple models with cross-validation.
3. **Optimization**: Fine-tunes hyperparameters using grid or random search.
4. **Ensemble**: Combines the strengths of multiple models, if desired.

PyCaret emphasizes ease of use and fast experimentation, making it suitable for users who need quick insights into their data without in-depth coding.

2.5.4.3 H2O

H2O is a scalable, open-source AutoML platform designed for large-scale ML tasks and big data applications [11]. It supports distributed, in-memory processing, allowing it to handle extensive datasets efficiently. H2O's AutoML module automatically trains and tunes a selection of models, including ensemble models, within a specified time frame or number of models.

H2O's model training can be formalized as:

$$\max f(\mathbf{X}, \mathbf{y}) = \sum_{i=1}^{n} \text{model}_i(\mathbf{X}, \mathbf{y}) \tag{2.55}$$

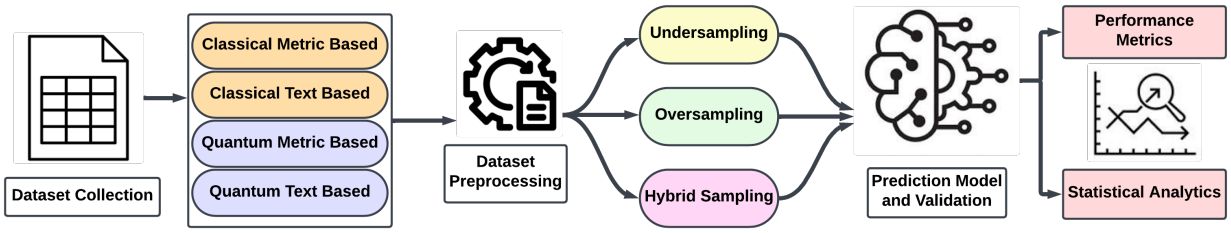


Figure 2.7: Flow of Experimental Design

where $\mathbf{X}$ and $\mathbf{y}$ are the input features and target values, and model_i represents individual models trained and tuned within the H2O AutoML pipeline. By leveraging stacking and ensemble methods, H2O can produce robust models that outperform single models in terms of predictive accuracy and generalization.

H2O also supports DL architectures and is commonly used in industries requiring scalable ML solutions, such as finance, healthcare, and marketing.

2.6 Experimental Design Framework

The experimental design of this thesis is illustrated in Figure 2.7, with the various steps outlined and explained in the subsequent sections.

2.6.1 Empirical Data Collection

For empirical validation, the majority of the data used in this study was sourced from publicly available datasets related to code smell detection. These datasets were primarily obtained from academic research papers that published their datasets for use in related studies. In addition, open-source software systems, which are continuously maintained and modified by global development communities, were also utilized. These software systems were selected for their relevance to common code smells such as *God Class*, *Long Method*, and *Feature Envy*.

The majority of the data was collected from open repositories hosted on GitHub and Zenodo, where these software systems are freely available. They offers a vast selection of repositories with both historical and recent code versions, making it an ideal platform for extracting diverse datasets. For this study, the source code from selected repositories was downloaded and analyzed to identify code smells using existing detection techniques. This provided a rich and varied dataset that includes different types of code smells, which was then structured for further experimentation and analysis. The open-source nature of these repositories ensures transparency and replicability in the data collection process.

2.6.2 Software Systems used for Data Collection

2.6.2.1 Classical Metric Based

The classical metric-based approach to code smell detection has been extensively explored using a wide range of datasets, each offering unique insights into different software projects and their associated code smells. These datasets provide a rich source of information, allowing for a comprehensive analysis of code smells based on various software metrics such as coupling, cohesion, complexity, and size. This study has employed multiple datasets, including Apache, JEdit, EDTforCSD, etc. These were some of the open source independent projects while there were some datasets that were combined using different projects and served as multiclass dataset [121].

- **jEdit:** This dataset originates from the jEdit open-source text editor and was first utilised in the work by Fontana et al. for empirical studies on code smells [122]. It includes 34 static code metrics per method, the JEdit text editor, a widely used open-source text editor written in Java and exhibits an imbalance ratio of 0.21. This dataset is suitable for evaluating detection performance in moderately imbalanced environments and offers insights into how code smells affect the maintainability of tools designed for developers, providing a unique perspective on the challenges of software quality in editing applications.

- **Apache:** Derived from multiple Apache software repositories, this dataset was introduced by Kershsavarz and Nagappan[123]. It includes 34 code metrics and provides a relatively larger and more balanced environment than jEdit, making it ideal for studying generalisation across mid-sized projects. The Apache dataset [123] is known for its extensive collection of open-source projects, such as Apache Ant [124], Tomcat [125], and Maven [126], which have been widely used in software quality research. The diversity and size of Apache projects offer a broad spectrum of metrics, making it a valuable dataset for studying various code smells.

- **EDTforCSD:** One of the largest and most severely imbalanced datasets in this study, EDTforCSD was introduced by Tufano et al. [127]. It includes 28 software metrics extracted from open-source repositories and presents an extreme imbalance ratio of 0.01. It serves as a rigorous test case for evaluating resampling strategies.

Additionally, Palomba et al. [121] used dataset that serves as a crucial resource in code smell detection research. This dataset, introduced in an empirical study by Palomba et al. [19], contains labelled instances of five distinct code smells collected from various versions of 30 Java open-source projects. The dataset collection process involved examining

395 releases across these projects, resulting in approximately 17,350 manually validated instances encompassing 13 different types of code smells. For this study, only the six most commonly known code smells were selected, which include Complex Class, Large Class, Lazy Class, Refused Bequest, and Spaghetti Code. This dataset is particularly valuable because it includes 21 diverse software metrics, such as coupling and cohesion, which are employed to assess the presence of the prevalent code smells. The comprehensiveness and manual validation of this dataset make it a benchmark for evaluating the performance of code smell detection models.

These metric-based datasets of classical programs have been instrumental in the development and evaluation of various code smell detection models, offering a robust foundation for feature extraction and analysis of relevant characteristics. By utilizing these datasets, researchers have effectively identified, compared, and refined detection techniques, thereby contributing significantly to the understanding and management of code smells in diverse software systems.

2.6.2.2 Classical Text Based

In this study, two widely recognized classical datasets, namely **MLCQ** and **DesigniteJava**, have been utilized for code smell detection tasks. These datasets provide textual and metric-based features derived from large-scale software systems and have been frequently employed in empirical software engineering research.

The **MLCQ** dataset, proposed by Arcelli Fontana *et al.* [128], was specifically designed to support ML models in the detection of multiple code smells. It consists of 15,000 instances extracted from 74 open-source Java projects, covering four widely studied code smells: *God Class*, *Feature Envy*, *Long Method*, and *Data Class*. Each instance is represented by 34 source code metrics that capture structural and behavioral properties such as coupling, cohesion, inheritance, and complexity. A distinguishing feature of MLCQ is its balanced inclusion of both smelly and non-smelly classes, making it suitable for evaluating classification models across multiple code smell types. The dataset is also labeled at the class and method levels, enabling multi-granularity analysis of code smell detection.

The second dataset, **DesigniteJava**, was introduced by Shatnawi and Li [129]. It was constructed using the Designite tool, which statically analyzes Java projects to detect design smells. The dataset includes 27,900 instances derived from a collection of open-source Java systems, covering a wide range of 19 different design smell types. Each instance is characterized by 35 metrics, including size, complexity, coupling, and inheritance-related measures, providing a rich representation of software structural properties. Unlike MLCQ, which focuses on a limited set of well-known smells, DesigniteJava offers a broader coverage of design smells, thereby enabling large-scale comparative evaluations

of different code smell categories. Due to its size and diversity, DesigniteJava has become a benchmark dataset for empirical investigations in design smell detection.

2.6.2.3 Quantum Metric based

Quantum metric-based code smell detection is an emerging research area that adapts principles of QC to assess design flaws in quantum programs. Early studies have proposed metrics capturing attributes such as gate count, circuit depth, and circuit complexity, which provide insights into maintainability and understandability of quantum software [130, 131]. For instance, Cruz-Lemus et al. [131] adapted classical software metrics to quantum circuits, emphasizing readability and modularity through measures like circuit depth and gate count.

Chen et al. [70] further advanced this domain by creating a dataset of 15 quantum programs, where eight quantum metrics (CG, ROC, NC, LC, IM, LPQ, IdQ, and IQ) were defined as thresholds to identify smelly characteristics. This dataset represents one of the first systematic efforts to capture quantum-specific code smells.

Despite these promising directions, the field remains underdeveloped. Current approaches mainly focus on structural complexity and readability, while challenges such as semantic analysis of entanglement, coherence, and superposition remain open. Future research should aim to design dedicated quantum smell metrics and empirically validate them for practical refactoring and optimization in hybrid classical–quantum systems.

2.6.2.4 Quantum Text Based

With the advent of quantum software engineering, new datasets have been introduced to capture the unique characteristics of quantum program code. Among them, **LintQ** and **Bugs4Q** stand out as representative text-based datasets that support empirical studies in quantum code analysis, defect prediction, and software quality research.

- **LintQ:** Introduced by Paltenghi et al. [132], **LintQ** is one of the largest collections of QP code snippets. It contains over 7,500 real-world projects gathered from open-source repositories and includes code in multiple formats such as Python scripts, Jupyter notebooks, text files, and executable files. The dataset spans major quantum programming frameworks, including Qiskit, Cirq, and TensorFlow Quantum, reflecting the diversity of current quantum development practices. LintQ captures snippets from different project stages and commits, providing a representative yet heterogeneous source of quantum code. Due to its raw nature, preprocessing is often required to normalize and clean the data for metric extraction or static analysis.

This dataset is particularly suitable for large-scale empirical studies on structural, semantic, and quality-related aspects of quantum programs.

- **Bugs4Q:** Proposed by Zhao et al. [133], **Bugs4Q** is a curated dataset focusing on reproducible bugs in QPs. Initially built from Qiskit repositories, it covers core components such as Terra, Aer, Aqua, and Ignis, with later extensions to include Cirq and Microsoft Q#. Each bug instance consists of buggy code, its associated patch, and relevant metadata, enabling reproducibility and validation. The dataset systematically categorizes bug types and provides linked test cases, making it a reliable benchmark for empirical studies in defect prediction, debugging, and testing of quantum software.

While LintQ provides a large-scale and diverse source of raw quantum code for exploratory studies, Bugs4Q offers a curated and reproducible benchmark for bug-related research. Together, they form complementary resources for advancing empirical research in quantum software engineering.

2.6.3 Data Preprocessing

Pre-processing is a critical step in preparing datasets for code smell detection, as it ensures the consistency, cleanliness, and relevance of data before it is fed into ML models. This section outlines the various techniques employed to transform raw datasets into a structured format suitable for classification and analysis.

- **Cleaning Code:** Removing comments, unnecessary whitespaces, and non-executable parts of the source code to ensure consistency.
- **Tokenization:** Splitting the source code into tokens such as keywords, operators, identifiers, and literals for structured analysis.
- **Padding:** Ensuring uniform input lengths by adding padding tokens so that models can process sequences consistently.
- **Handling Missing Values and Transforming Features:** Addressing incomplete data by imputation or removal and applying transformations such as normalization or encoding to prepare features for learning.
- **Format Standardization:** Converting data into a uniform format (e.g., CSV, JSON, or tensor form) suitable for machine learning pipelines.

2.6.4 Data Balancing

In this thesis, sampling techniques were employed to address the issue of imbalanced data distribution, including (i) oversampling, (ii) undersampling, and (iii) hybrid sampling. These data balancing techniques help mitigate class imbalance by adjusting the dataset's composition before model training. In oversampling, additional instances are generated for the minority class, either through simple replication or by creating synthetic examples, such as with SMOTE, ensuring that the model learns adequately from all classes. In contrast, undersampling reduces the number of majority class instances by selecting a representative subset, preventing the model from disproportionately focusing on the majority class, though it may risk information loss. Hybrid sampling combines the advantages of both oversampling and undersampling, producing a more balanced dataset with minimized risk of bias and data loss. Detailed explanations of these data-balancing techniques are provided in Chapter 4.

2.6.5 Prediction Model and Validation

In the present study, ML models were developed to predict the presence of code smells within the selected datasets. The process of model construction involved careful selection of classifiers, feature representation, and evaluation strategies. The prediction model was trained on preprocessed data where relevant features were extracted and transformed to ensure compatibility with the chosen algorithms. Both classical and advanced models were employed, and their predictive capabilities were systematically assessed.

To ensure the robustness and generalisability of the results, appropriate validation strategies were implemented. Among the commonly adopted methods, *k*-fold cross-validation was selected due to its reliability in reducing bias and variance in model evaluation [134]. In this approach, the dataset is partitioned into k equally sized folds where k = 10. The model is trained on $k-1$ folds and validated on the remaining fold, repeating the process k times as illustratied in Figure 2.8. The final performance is reported as the average across all folds, which provides a more stable estimate than a single train-test split. This procedure is particularly suitable for datasets where imbalance or limited sample size could otherwise lead to unreliable estimates. Mathematically, the average cross-validation score can be expressed as:

$$\text{CV Score} = \frac{1}{k}\sum_{i=1}^{k}\text{Accuracy}_i \tag{2.56}$$

where $k = 10$ in ten-fold cross-validation, and Accuracy_i represents the accuracy obtained on the i-th fold. This approach balances model training and testing, making it ideal

for small to medium-sized datasets where test data scarcity might be an issue.

While cross-validation techniques measure the predictive accuracy and generalisation ability of the models, they do not explain how predictions are derived. To address this, model interpretability was examined through XAI methods, specifically SHAP and LIME.

- **SHAP**: SHAP values are grounded in Shapley values from cooperative game theory, providing a unified approach to attribute the prediction output of a model to its input features [135]. For a given prediction, SHAP calculates the contribution of each feature by considering all possible feature combinations. The SHAP value for a feature x_j is given by:

$$\phi_j = \sum_{S \subseteq \{1,\ldots,n\} \setminus \{j\}} \frac{|S|!\,(n - |S| - 1)!}{n!} \Big(f(S \cup \{j\}) - f(S) \Big) \tag{2.57}$$

  where S represents a subset of all features excluding j, and $f(S)$ is the model's prediction when only features in S are considered. SHAP values provide a global view of feature importance and are especially useful for understanding non-linear interactions in complex models.

- **LIME**: LIME provides interpretability by approximating the model locally around a specific prediction [136]. LIME generates perturbed samples around the instance being explained and observes the changes in prediction. It then fits a simpler, interpretable model (like linear regression) to approximate the predictions in this local region. The interpretability is provided by the coefficients of the locally-fitted model, indicating the effect of each feature near the instance in question. The LIME explanation for a prediction $f(x)$ can be formalized as:

$$\hat{f}(x) = \arg\min_{g \in G} \; L(f, g, \pi_x) + \Omega(g) \tag{2.58}$$

  where L measures the fidelity of g (the local surrogate model) to f (the original model), π_x denotes locality around x, and $\Omega(g)$ enforces simplicity in g. LIME is particularly useful for providing insight into individual predictions in black-box models.

Together, the use of cross-validation for validation and XAI techniques for interpretability establishes a balanced framework. This not only confirms that the prediction models perform reliably but also provides transparency into their decision-making process, which is crucial in domains where software quality and maintainability are being assessed.

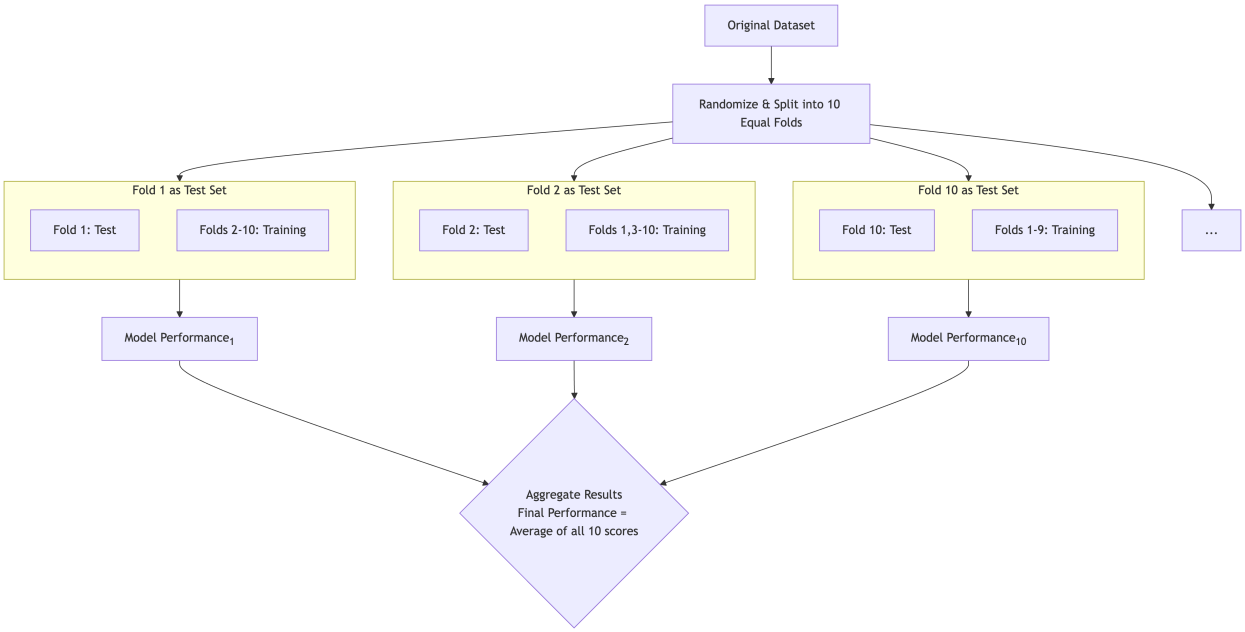


Figure 2.8: Illustration of Ten-Fold Cross Validation

2.6.6 Performance Measures

To evaluate the effectiveness of the developed prediction models, a set of well-established performance measures is employed. These measures allow a fair comparison across different models and help in understanding their strengths and weaknesses under imbalanced and real-world datasets. Each metric has specific strengths and helps capture different aspects of model effectiveness, particularly for imbalanced datasets or multi-class classification problems. The key metrics used in this study are Accuracy, Precision, Recall, F1-score, MCC, and AUC.

2.6.6.1 Accuracy

Accuracy is one of the most straightforward evaluation metrics for classifiers. It measures the ratio of correctly predicted instances to the total number of samples. Although accuracy provides an overall view of performance, it can give a biased impression in cases where the dataset is highly imbalanced. The formula for accuracy is expressed as:

$$\text{Accuracy (Acc)} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.59}$$

where TP denotes true positives, TN represents true negatives, FP corresponds to false positives, and FN indicates false negatives.

2.6.6.2 Precision (P)

Precision measures the proportion of true positive predictions among all positive predictions made by the model. It reflects how reliable the model is when it claims the presence of a code smell. It is useful for scenarios where false positives are costly. Precision is calculated as:

$$\text{Precision (P)} = \frac{TP}{TP + FP} \tag{2.60}$$

A high precision indicates that the model makes fewer false positive errors.

2.6.6.3 Recall

Recall, also known as sensitivity or true positive rate, measures the proportion of actual positives that were correctly identified by the model. ndicates the ability of the model to identify actual positive instances. Recall is given by:

$$\text{Recall (R)} = \frac{TP}{TP + FN} \tag{2.61}$$

High recall ensures that most of the defective or smelly cases are detected, which is crucial in software quality tasks.

2.6.6.4 F1-Score (F1)

The F1-score is the harmonic mean of precision and recall, providing a balance between these two metrics. It is useful when there is an uneven class distribution, as it penalizes extreme values in either precision or recall. The F1-score is computed as:

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.62}$$

A high F1-score indicates both high precision and high recall, making it a suitable metric for imbalanced datasets, as it balances the trade-off between P and R.

2.6.6.5 Positive Rate

The positive rate is a performance measure that tells how often the model (or test) predicts the positive class relative to all instances. It's usually expressed as:

$$\text{Positive Rate} = \frac{TP + FP}{TP + TN + FP + FN} \tag{2.63}$$

It expresses the fraction of all cases that are labeled as positive by the model, whether correctly or incorrectly. It is useful for checking if a classifier is biased toward predicting one class more often.

2.6.6.6 Matthews Correlation Coefficient (MCC)

MCC is a comprehensive evaluation metric that incorporates all four outcomes of a confusion matrix: true positives, true negatives, false positives, and false negatives. Unlike accuracy, it remains reliable even in scenarios with class imbalance. The score ranges from -1 to $+1$, where $+1$ reflects perfect agreement between predictions and ground truth, 0 corresponds to performance equivalent to random guessing, and -1 indicates complete disagreement. The formula for MCC is given as:

$$\text{MCC} = \frac{(TP \cdot TN) - (FP \cdot FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{2.64}$$

By combining information from all components of the confusion matrix, MCC provides a more balanced evaluation of model performance, particularly in cases where dataset imbalance might render accuracy misleading.

2.6.6.7 Area Under the ROC Curve (AUC)

AUC represents the area under the ROC curve, which plots the true positive rate (recall) against the false positive rate. AUC values range from 0 to 1, with higher values indicating better model discrimination between positive and negative classes. AUC is especially useful in binary classification and can provide insights into model performance across various threshold levels. It is valuable for understanding the model's discriminative power between positive and negative classes.

Together, these performance measures provide a comprehensive picture of model behavior, ensuring that evaluation goes beyond simple correctness and captures reliability, sensitivity, and generalization capabilities.

2.6.7 Statistical Analysis

Statistical analysis is essential in evaluating the performance of models and determining whether observed differences in performance metrics are statistically significant. In this study, we employ the Friedman and Wilcoxon tests to compare the performance of multiple models across different datasets.

2.6.7.1 Friedman Test

The Friedman test is a non-parametric statistical test used to detect differences in treatments across multiple test attempts. It is particularly useful when comparing three or more algorithms or models across multiple datasets or conditions, as it ranks the performance and determines if there is a statistically significant difference among the models. The Friedman test is an extension of the Wilcoxon signed-rank test for more than two groups, assuming no normal distribution.

Given k treatments (models) and n subjects (datasets), the test statistic Q is calculated as:

$$Q = \frac{12}{nk(k+1)} \sum_{j=1}^{k} R_j^2 - 3n(k+1) \tag{2.65}$$

where R_j is the sum of ranks assigned to the j-th model across all datasets. The test statistic Q approximately follows a chi-squared distribution with $k-1$ degrees of freedom.

If Q exceeds the critical value from the chi-squared distribution, we reject the null hypothesis, indicating that at least one model performs differently from the others.

2.6.7.2 Wilcoxon Signed-Rank Test

When the Friedman test indicates significant differences among multiple models, post-hoc pairwise comparisons can be performed using the Wilcoxon Signed-Rank test. This non-parametric test evaluates whether there is a statistically significant difference between two related samples, making it a suitable alternative to the paired *t*-test when the assumption of normality is violated. In the context of model comparison, it is often used to test whether two algorithms perform differently on the same datasets.

To compute the test statistic, let d_i denote the difference in performance between two models on the ith dataset. All zero differences are discarded, and the absolute differences $|d_i|$ are ranked. The ranks are then assigned the sign of their corresponding d_i. The test statistic W is defined as:

$$W = \min\left(\sum_{d_i>0} \text{rank}(|d_i|), \sum_{d_i<0} \text{rank}(|d_i|)\right) \tag{2.66}$$

The null hypothesis states that there is no difference between the two models. For larger sample sizes, the distribution of W can be approximated by a normal distribution with mean and variance:

$$\mu_W = \frac{n(n+1)}{4}, \quad \sigma_W = \sqrt{\frac{n(n+1)(2n+1)}{24}} \tag{2.67}$$

and the standardized test statistic is given by:

$$Z = \frac{W - \mu_W}{\sigma_W} \tag{2.68}$$

If the obtained Z-value falls in the critical region, we reject the null hypothesis and conclude that the two models differ significantly in performance.

Thus, while the Friedman test provides an overall indication of differences across multiple models, the Wilcoxon Signed-Rank test enables rigorous pairwise analysis, ensuring that performance differences are statistically meaningful.

Chapter 3

Systematic Literature Review

This chapter presents a SLR on the application of DL techniques for code smell detection. Covering studies published between January 2013 and July 2023, the review focuses on four key aspects: the types of code smells addressed, the training methods used, the role of software metrics in evaluation, and the performance of proposed models. By consolidating recent findings, the chapter highlights current progress, identifies research gaps, and outlines directions that can support the development of more effective and reliable detection approaches. The remainder of this chapter is structured as follows: Section 3.1 presents the background relevant to the review, while Section 3.2 outlines the overall SLR process. The findings are reported in Section 3.3, and the chapter concludes with a discussion and future research directions in Section 3.4. The work presented in this chapter has been published in Cluster Computing Journal, Springer [137].

3.1 Background

Software maintenance is one of the most demanding stages of the software development life cycle. Once software is deployed, it often needs to be modified to fix bugs, add new features, or adapt to changing requirements [138]. In practice, these changes are not always carried out under ideal conditions. Tight deadlines and the pressure to deliver quickly can result in shortcuts or poor coding practices, which accumulate over time as technical debt. This debt often surfaces in the form of antipatterns or smells at the code or design level [139, 140]. Among these, code smells have received particular attention because of their impact on everyday development work [17, 141]. They do not typically break a system immediately, but they make code harder to read [142], more error-prone [143, 144], and costlier to maintain [145, 146]. Left unchecked, they can grow to a point where the software becomes fragile, difficult to extend, and in some cases, unmaintainable. Early detection of code smells was largely manual, relying on the experience of de-

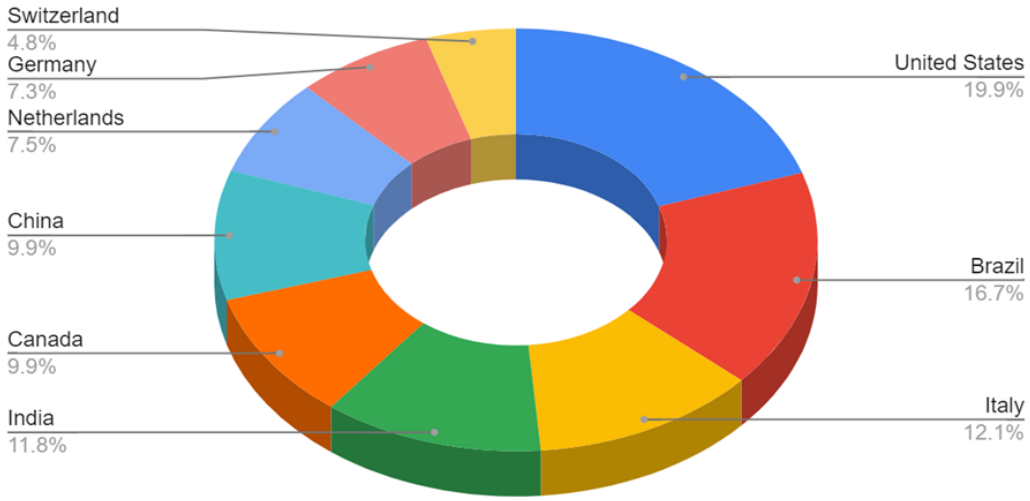


Figure 3.1: Contribution of Top 9 Active Countries in Code Smell Research

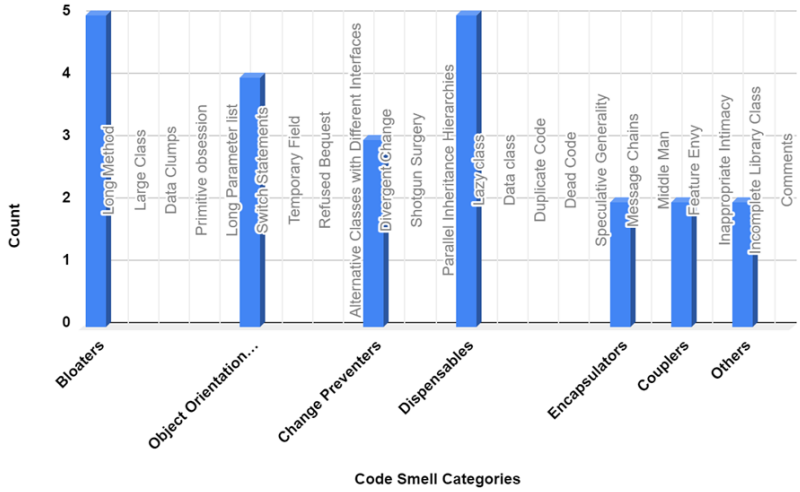


Figure 3.2: Code Smell Categorization given by Mantyla

Table 3.1: Code Smells Identified by Wake in 2003

Code Smell	Description
Complicated Boolean Expressions	Snippets of code having complex conditions involving operators like NOT, OR, and AND.
Dead Code	Having a variable, attribute, or code not utilized elsewhere. It typically happens after a code update with a poor cleanup.
Inconsistent Names	The same term is used in several contexts and for various reasons.
Null Check	Repeatedly occurring events, checking the object's null values
Magic Numbers	Numbers that are purposefully used in the code and whose values never change.
Special Case	Lots of complex conditional statements are used.
Type Embedded in Name	This includes commonly used names with redundant definitions like "course.addSubject(subject)" naturally instead of course.add(subject). The usage of Hungarian notation, whose types are opposite to their functions or purposes, was also included in this category.
Uncommunicative Names	Names given to software items, often local variables and attributes, need to be more descriptive of their names or intended purposes, such as y or value 0. When applied to methods and classes, it becomes much more crucial.

velopers. While useful, this approach was time-consuming and inconsistent. To address these challenges, researchers have increasingly turned to automated detection techniques. Over the years, contributions have come from many parts of the world, with a noticeable concentration from the United States. The contribution from the top 9 active countries in this domain is shown in Figure 3.1. More recently, advances in DL have opened new opportunities for smell detection. However, challenges remain that many studies propose isolated solutions without comparing or building upon earlier work [21, 147], and evaluation frameworks are often limited, despite the growing complexity of modern software systems.

Figure 3.2 shows how code smells can be categorized, while Table 2.1 and 3.1 highlight some of the most widely recognized examples from previous research. Together, these insights emphasize the need for deeper investigation into DL-based detection methods that can improve software maintainability and help manage the long-term risks of technical debt.

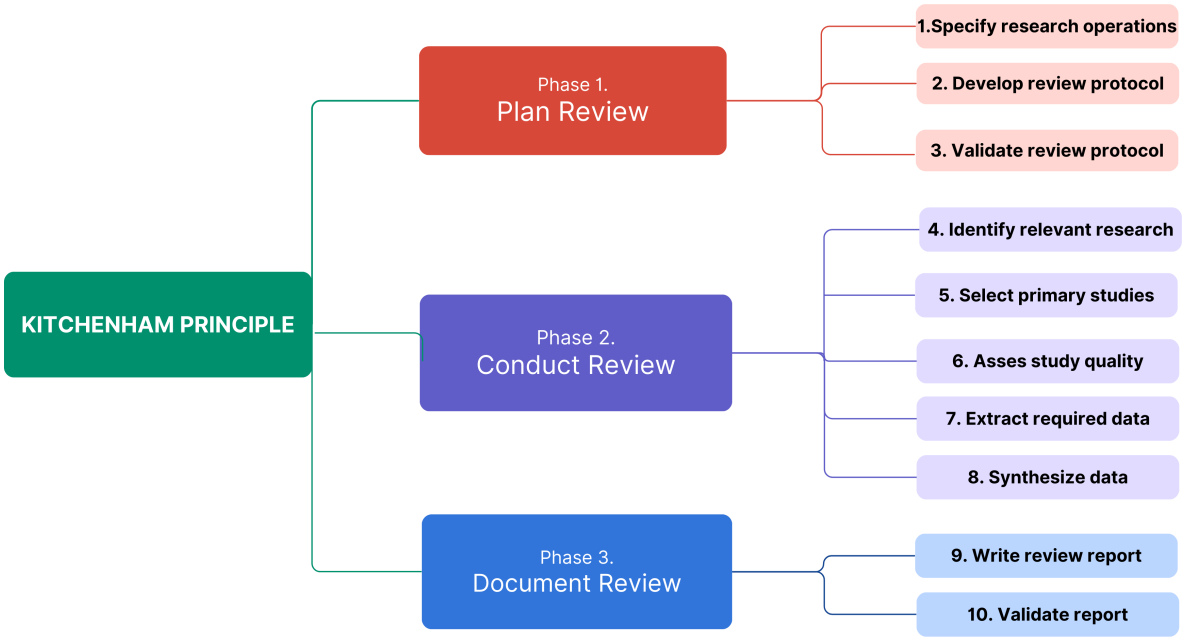


Figure 3.3: Kitchenham Strategy for SLR

3.2 Overview of the SLR Process

The following section contains the research strategy to analyze the research papers selected for this SLR. It is a structured and rigorous process for locating, evaluating, and examining published literature to examine a particular research issue or phenomenon. In 2007, Kitchenham [148] wrote a paper describing the guidelines for performing systematic literature reviews in software engineering. On the policies of Kitchenhem rules, this section is elaborated for planning, conducting, and reporting the study in a recurring order. All the phases and corresponding steps that should be taken for a systematic literature review according to Kitchenham principles are shown in Figure 3.3.

3.2.1 Planning

In the first stage, known as the planning stage, the search databases were found, Research questions were created, and the pertinent articles on the research topic were retrieved.

3.2.2 Research Questions

The primary goal of this comprehensive literature review is to highlight the range of code smells in software systems, their identification, and various code smell detection approaches. Table 3.2 describes the research questions formulated for this systematic review.

Table 3.2: Research Questions Imposed on this Systematic Literature Review

Research Questions	Motivation
RQ 1: Code smells considered	
RQ 1.1: What are the code smells currently detected by DL algorithms?	To investigate the most recent advances in DL-based code smell detection by identifying the code smells which have been considered to date.
RQ 2: Deep Learning Approaches	
RQ 2.1: Which independent variables give the best result in predicting code smells?	This will help to determine how many variables are necessary for the detection process.
RQ 2.2: Which dependent variable is used to get the results and analysis of code smells?	What data type will be the model output, and how can it be enhanced to get more accurate results?
RQ 2.3: Which DL algorithms are utilized in code smell detection?	This will help developers and researchers choose the most efficient DL algorithm for code smell detection.
RQ 2.4: What detecting tools are most commonly used in research for code smells?	In this strategy, many tools are used to utilize time properly. Investigating these tools can help to innovate more tools.
RQ 2.5: What strategies have been adopted for training the models?	Training in the best way possible will give the best results.
RQ 3: Evaluation Methods	
RQ 3.1: What datasets are used to detect code smells?	This will help to identify and create an accurately labeled dataset.
RQ 3.2: Which result metric is used to evaluate the models?	This will help researchers to compare the results of each tool or algorithm to determine if their metrics are similar.
RQ 4: Meta-analysis of Performance	
RQ 4.1: How do independent variables impact performance?	To know if the result varies with the count of variables, making it easier for tools to analyze the dataset.
RQ 4.2: Is the performance of code smell prediction models affected by DL algorithms?	How many layer(s) gives the best output and help the model to perform better by trying new combinations and learning from past researchers.
RQ 4.3: What is the impact of Training Strategies?	Does within-project training and cross-project training show a significant difference? This will help to know how many datasets are needed to be compiled.



Figure 3.4: Related Terms for the Review

3.2.3 Search Strategy

A search method was developed to gather all the published material pertinent to this issue. The search approach included selecting search terms, criteria for selecting articles for the research, a search method, and the relevant sources for articles.

3.2.3.1 Search Term Used

To determine the appropriate search terms, a strategy was followed:-

1. Firstly, by identifying the population, the intervention, and the result, a list of key phrases was derived from the study questions as shown in Figure 3.4.

   Result: Regarding the initial stage, the focus was on identifying and structuring the critical terms for the population, intervention, and outcome. The population encompasses code smells and anti-patterns that manifest within the code of software systems. The interventions involve a range of DL techniques, algorithms, tools, and datasets designed for detecting, recognizing, and prognosing code smells. The outcomes encompass a variety of identified code smells through the application of DL algorithms, detection tools, and the metrics employed within the detection process.

2. Alternate spellings and synonyms for all significant terms were looked upon.

   Result:

- *Code Smell:*“code smells” OR “code smell” OR “code bad smells” OR “bad code smells” OR “bad smell” OR “anomalies” OR “antipatterns” OR “antipattern” OR “design defect” OR “design-smells” OR “code defect” OR “code flaw” OR “smelly code” OR “design flaw”
- *Deep Learning:* "Deep Learning" OR "Neural Networks" OR "Artificial Neural Networks" OR "Deep Models" OR "Layered Neural Networks" OR "Multi-Layer Perceptron" OR "Deep Architectures" OR " Convolution Neural Network" OR "Hybrid Model" OR "CNN" OR "DL" OR "ANN" OR "AST" OR "RNN"
- *Prediction:* (”prediction” OR ”detection” OR ”identification” OR ”prediction model” OR ”model”
- *Software:* "Software Maintenance" OR "Software Engineering"

3. Keywords were checked in pertinent papers, and keywords unrelated to code smells were eliminated.

   Result: eliminated - ”anomalies” OR ”antipatterns” OR ”design defect” OR ”design-smells” OR ”design flaw” OR "code bad smells" OR "bad code smells"

4. For the conjunction of two or more cases, AND and OR boolean operators were used, wherein OR operator was used for the concatenation of alternative spellings and synonyms. In contrast, the AND operator was used to concatenate major terms.

5. Lastly, the search string was integrated into the summarized form. The final Search term is mentioned below.

(Software Engineering OR software maintenance) AND (antipattern OR code smell OR bad smell OR code defect OR code flaw OR smelly code OR anti-pattern) AND (deep learning OR CNN OR DL OR ANN OR AST OR RNN) AND (identification OR detection) AND (tool OR method OR technique OR approach)

3.2.3.2 Databases to be Searched

A significant part of an SLR is the choice of appropriate resources to use while looking for relevant material. To find all the material pertinent to this study’s concerns, the following resources were used: Springer, Google Scholar, Wiley, IEEE Xplore, Elsevier Science Direct, ACM, and Engineering Village.

The desire to compile as many papers as possible to adequately carry out this systematic literature review motivated us to choose these databases. Since these sites have a vast

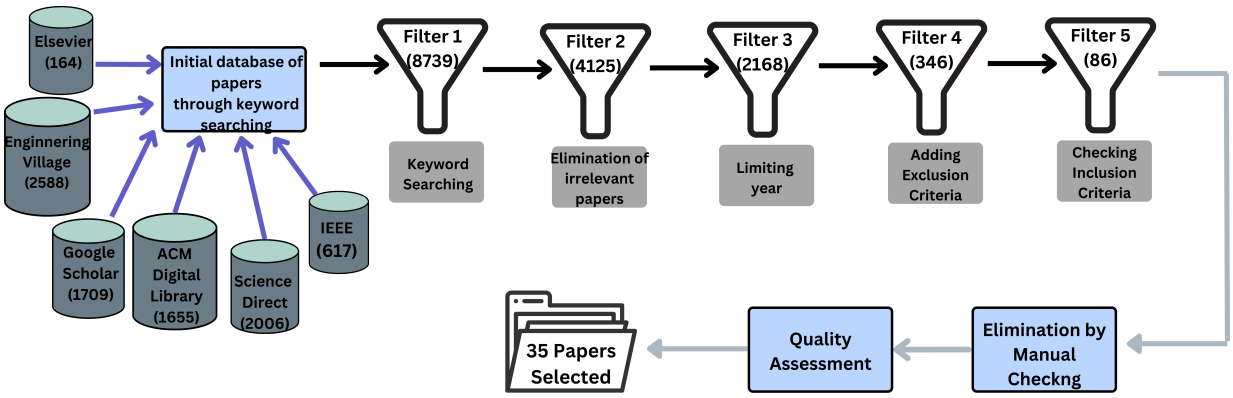


Figure 3.5: Article Selection Process

volume of literature - conference proceedings, books, journal articles, etc. related to the paper's objectives, the sources analyzed are regarded as the most representative of code smell detection and are utilized in different SLRs.

3.2.4 Article Selection Process

This section will discuss the criteria for narrowing down the number of research articles examined. The method used in this study to select primary studies is shown in Figure 3.5. The selection procedure is covered in depth in the following subsections:-

3.2.4.1 Databases and Search Terms Used

The study followed four main steps to examine and eliminate papers irrelevant to this study and select which are useful to the research questions:

1. Using the search terms listed in section 3.2.3.1 and shown in Figure 3.4, this study gathered primary research from the online libraries described in section 3.2.3.2. No time limit was set for the search procedure so that enough relevant materials could be gathered. Table 3.3 displays the search results from the digital libraries.

2. In this search criteria, keyword searching played a vital role involving search engines or databases to identify articles, books, or other materials that match specified keywords or phrases after the keyword research stage. Second, this study used citation searching, or, more precisely, snowballing, to review these papers. The "snowball" strategy is used for searching for potentially lost papers. In particular, "snowballing" involves finding new sources by consulting a paper's references or citations. Both forward and backward snowballing were utilized in this context; the initial one has all the papers cited in the first chosen articles, while the latter has all

Table 3.3: Layer-wise Article Selection Process

Sources	Filter 1	Filter 2	Filter 3	Filter 4	Filter 5	Final Filter
IEEE	617	232	156	20	8	4
Elsevier	164	159	68	21	9	4
Science Direct	2006	732	475	84	13	6
Engineering Village	2588	1311	477	85	26	11
Google Scholar	1709	707	451	89	19	7
ACM	1655	984	541	47	11	3
Total	8739	4125	2168	346	86	35

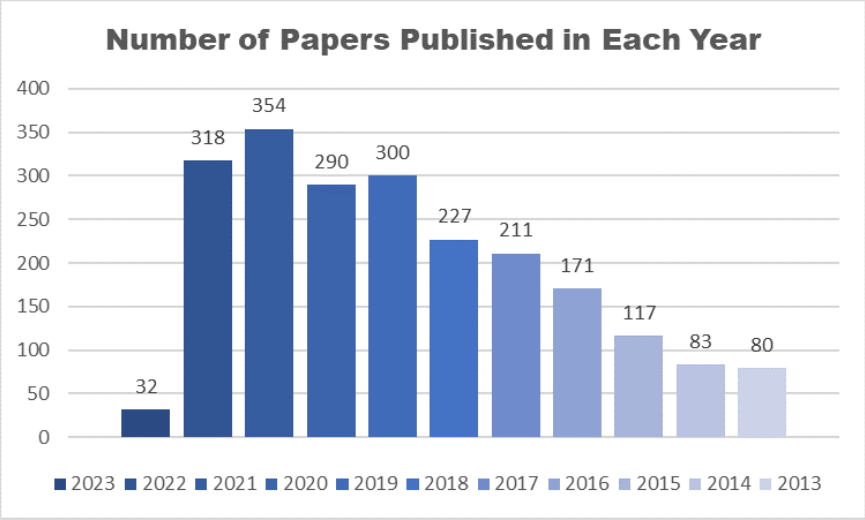


Figure 3.6: Year-wise Publication in Last Ten Years

papers that cited the initially selected papers. These various search methods gave a raw database of 8739 publications, which satisfied the first filter.

3. In the second step, all the irrelevant papers were eliminated by checking their relevancy to the topic and the depth of their research. The third layer involved selecting all the papers which were published after 2013. Figure 3.6 gives the number of papers year-wise written from 2013 till date.

4. For the fourth filter, an exclusion criterion was explicitly designed to match the relevant research questions. The criteria firstly remove all the papers irrelevant to the title of this research. This eliminated unrelated publications from the whole list of retrieved sources using the exclusion criteria (explained in Section 3.2.4.2). This stage was carried out by the second author of this research, who examined the 2,168 publications that were located, applied the exclusion criteria, and read their titles, abstracts, and keywords. The articles that did not satisfy these requirements were eliminated, and the remaining papers were picked for the inclusion criteria, which comprised the fifth screening step.

5. Similar to how the exclusion criteria were used, the inclusion criteria were used as the next filter. The databases that met this requirement were preserved, while the others were deleted. Each paper was evaluated against the inclusion criteria by the second author of this work, after which these papers underwent additional filtering.

6. After the automatic detection and removal of irrelevant papers, the authors eventually turned to manual validation and verification to eliminate duplicates, irrelevant papers, or articles that didn't satisfy the paper's requirements. For this elimination phase, all three writers worked to assess the relevant portions of all the publications. From this method, the final search string selection comprised 86 articles (or 0.6 percent of the papers discovered in the previous stage).

7. The 54 pertinent software engineering conferences and 32 journals were all searched manually to ensure all relevant literature was included in the ending filter and added to the final list of chosen sources. More specifically, depending on the exclusion/inclusion criteria, we reviewed each of the paper sources and their relevancy that were discovered before deciding whether or not to include them in the final selection or not. To achieve this, she started by looking at the title. If it was obviously outside the scope of the study (for example, if it mentioned automatic test case generation), it was decided to skip the paper. In contrast, if a publication was seen as a potential paper, we read the abstract, introduction, and conclusion to confirm its applicability. Finally, 35 studies were chosen from this filter stage.

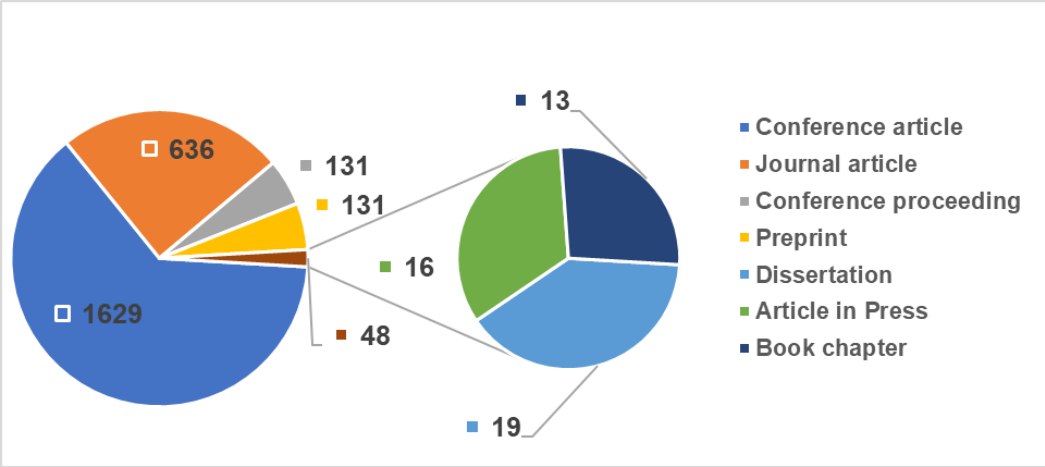


Figure 3.7: Article Published in various categories

8. With the final set of papers, we did a last recheck, i.e., the quality assessment phase, to double-check if they have relevance to the topic. Table 3.4 describes the quality assessment criteria formulated for this research. This step resulted in the final set of papers examined in this research. All 35 sources discovered up to this point passed the quality assessment. Section 3.2.4.4 describes the data extraction procedure.

9. After this long examination, including keyword searching, snowballing, and manual search, **35** papers were selected and surpassed all tests. Table A.1 describes the list of all 35 selected studies with their titles.

3.2.4.2 Inclusion and Exclusion Criteria

A paper has to meet the following criteria to be useful in explaining the RQs.

1. **Exclusion criteria:** For all the layers, the authors put up some criteria for excluding research papers from the total found. Sources that complied with the conditions listed below were disregarded for this analysis:

    - Articles with less emphasis on DL and more on other detecting methods -: other methods like machine learning and manual detection should be removed for effective verification of DL techniques used;
    - Articles authored in languages other than English -: To ensure resource efficiency and language proficiency, a broader scope is adopted;
    - Articles for which no full text is available -Excluding articles that lack open access, are in progress, consist of abstracts or theses, and eliminate duplicate entries for in-depth examination.

Table 3.4: Questions for Quality Assessment

Parameter	Question
Aim	Is the aim of the study specified and relevant to the review?
State of the art	Did the authors compare the results with the State of the art technology?
Dataset	Are the datasets of the corresponding study available publicly?
Accuracy	Can the provided approach accurately identify the code smells?
Variables	What are the precise independent and dependent variables required for the detection of particular code smell ?
Algorithm	Is the DL classifier's definition clear?
Evaluation metrics	Are the evaluation metrics and procedures explicitly disclosed?
Improvement	Are the improvements in quality attributes specified?
Limitations	Are the limitations of the papers explicitly discussed?

2. **Inclusion criteria:** Paper with significance to DL were to be kept safe from getting lost; hence, apart from excluding sources that complied with the requirements listed above used in this study, a list of inclusion criteria was framed as well:

    - Papers that provide publicly accessible source code and utilize openly available Java projects:- This will enhance the granularity of the dataset assessment process;
    - Articles that present novel methods for enhancing the effectiveness of DL methods currently in use for code smell detection:- Developed a novel approach and described the new method and model;

    The study included all kinds of articles (e.g., journals, proceedings, and conferences) to compile the complete selection of relevant sources. Figure 3.7 shows the number of articles in various categories.

3.2.4.3 Quality Assessment

After the last phase of selection, the papers' quality was assessed. The mentioned list ensured the publications' integrity and thoroughness choices. Answers of "Yes," "Partially," or "No" were given to each of the questions mentioned in Table 3.4. The study was con-

Table 3.5: Data Extraction Form

Dimension	**Attribute: Description**
Variety of code smells detected	Classification of various code smells being detected in the analyzed papers.
Programming Language	The technique used in the paper is an application to which programming language, e.g., Java or C++, etc.
DL Algorithms	Which kind and how many layers of each are employed in the study?
Type of Classification	What form of classification, such as multi-valued classification, binary classification, etc., was used to tell the code smell detection result?
Training approach	Which type of internal or external training approach was used?
Evaluation Metrics	What assessment measures, such as Precision, AUC-ROC, Recall, and F Measure, have been used to evaluate the model's accuracy?
Datasets	Which collection of data was utilized for training the model? A list of the software projects to which approaches for code smell detection have been used.
Independent Variables	The indicators used to assess how likely a code component is to be impacted by a code smell, such as LCOM, LOC, etc.
Dependent Variables	The indicator of a source code component's smelliness, such as the class level, method level, etc.
Limitations	What research gaps are present in the current techniques?
Future Work	What are the future scopes of these research papers that can be utilized for future research?

sidered lacking when a research's methodological components could have been deduced from the text, even if they were not clearly stated.

All the authors carried out the approach; they evaluated each paper source following the quality assessment criteria. All three authors simultaneously gave framed answers for each paper on their level. Then, the authors had a mutual discussion to approve all the articles.

This stage was moved through in rounds. Each time a paper was chosen, the researchers' responses to the various quality questions were compared. If there was a discrepancy, the participants had to defend their positions before another round could begin. Up till an agreement was reached, this procedure was repeated.

3.2.4.4 Data Extraction

The data required to address the research questions were extracted after choosing the final publications to be included in the SLR. Table 3.5 describes the data extraction form. In the data extraction, one column for research gaps was kept in addition to information regarding the specific features under consideration, such as the sorts of code smells taken into account in the chosen publications. As a result, it was easier for new research to understand the limitations of the previous ones. The authors of this work extracted the variables or attributes required to address this study's questions. However, all writers reasoned and examined each manuscript to identify potential limits before filling out the limitation section. All authors then combined the individual considerations.

3.3 Results of the Review

Table A.2 displays the final list of primary studies examined for this SLR, and the columns "Year" and "Publication Type" show how many articles were published in this domain over time in journals and conferences. As can be seen, all of the documents considered were published between 2013 and 2023 (till Date), out of which more than 65% of these primary studies were written after 2019, showing a developing trend in adopting DL approaches. Furthermore, it can be noted that most publications were journals; other than this, they also included conference papers and proceedings. This data indicates that most published articles have reported initial findings on how DL is used for code smell detection.

After taking a closer look at the writers of the papers, it was found that nearly 60% of these were co-authored, although the names and groups of such scientists were quite distinct. Some familiar authors, including Zhang, Liu, and Tushar Sharma, worked on different papers, but the number of papers published by the team of researchers, including them, was the most. Institutions like Athens University of Economics & Business that facilitate the studies for these researchers can be a guide to the upcoming scholars.

As a result, analysis of the primary studies determined the number of articles, the study areas researchers have covered, and the areas that still require more investigation. Young and new researchers might utilize this to get their study going in the direction the field dictates.

In Summary. It can be seen that there may still be unresolved issues with using DL algorithms for code smell detection based on the quantity and variety of research articles written by the community. Additionally, three different research teams were highlighted that are extensively working in this area of antipattern detection.

Table 3.6: List of Code Smells Detected in the Papers Analyzed under this SLR

S.no.	Code Smell Detected	Freq	References
1	Blob	1	[149]
2	Brain class and Brain method	2	[46],[150]
3	Complex Class	4	[151],[141],[45],[152]
4	Complex Method	1	[45]
5	Data Class	3	[141],[50],[43]
6	Empty Catch Block	3	[151],[45],[153]
7	Feature Envy	10	[154],[155],[156],[141],[157],[50],[153],[155],[43],[158]
8	God Class	7	[159],[141],[50],[153],[158]
9	Large Class	4	[160],[154],[157],[161]
10	Long Identifier	1	[45]
11	Long Method	5	[157] ,[45],[50],[153],[25]
12	Long Parameter List	2	[45],[152]
13	Magic Number	2	[151],[45]
14	Misplaced Class	1	[160],[157]
15	Multifaceted Abstraction	2	[151],[152]
16	Spaghetti code	1	[162]
17	Speculative Generality	1	[162]
18	Swiss Army Knife	1	[149]
19	Temporary Field	1	[163]
20	Others	6	[149],[164],[165],[153],[166],[43]

3.3.1 Results Specific to RQ1

This section examines a variety of code smells identified so far by DL models, giving insights into the most significant and least significant code smells.

3.3.1.1 RQ 1.1 - What are the code smells currently detected by DL algorithms?

This SLR shows that existing studies focused on more than 20 code smells listed in Table 3.6, with their occurrence frequencies. According to the data, 40% of studies focused on *Feature Envy*, which can also be stated to be the most frequently discovered code smell. With about 28% of work done on it, *God Class* is the second most popular smell. This result is partly expected because it is one of the most problematic and familiar code smells in development projects and has often garnered the most attention from researchers. Other smells that are widely considered in addition to them are the Long method (5), Large Class (4), Complex Class (4), Empty catch block (3), Data class (3), Long parameter list(2), Multifaceted abstraction (2), and Magic number (2) and Duplicate Code.

Spaghetti Code, Speculative Generality, Blobs, Misplaced Classes, and Temporary Fields are the less familiar smells in the code. It can also be highlighted that several unique code smells piqued the curiosity of scholars. As shown in Table 3.6, some publications identified the two most uncommon code smells, *Brain method* and *Brain class*. But other code smells like dead code, inappropriate intimacy, etc., have not been detected yet and require more attention from the researchers as they can cause significant harm to the software in the long run.

The analysis of the primary studies notably revealed an intriguing finding: only one approach could detect nine different code smells simultaneously. At the same time, other systems were restricted to identifying a maximum of one type of smell. This implies that developers are currently limited to DL methods that can handle only a restricted range of code smells.

Moreover, most previous research on prediction models for code smell identification relied heavily on automatic detection tools. Consequently, the availability of these detectors influenced the code smell detection being investigated. As a result, the emphasis in the field of antipattern study has primarily been on identifying the most prevalent code smells, likely due to the research methodology employed. However, more recent articles have started to shift their focus towards unique code smells, thereby expanding the scope of DL models for code smell identification.

Summary for RQ 1. DL-based detection primarily targets the smell including *Long Method, God class, and Feature Envy*. This leads to the further conclusion that widely used DL algorithms focus on the most typical smells. Apart from this, it is evident that research is taking a turn toward detecting new and rare code smells. Hence, the scope and area of research in code smell are widening.

3.3.2 Results Specific to RQ2

This SLR focuses on the DL setup employed in the literature. The paper analyzes the primary studies on the following parameters: (i) the independent variable, (ii) the dependent variables, (iii) the classifiers employed, (iv) the tools used, and (v) the training procedures used. The following subsections provide descriptions of the findings of this analysis.

3.3.2.1 RQ 2.1 - Which independent variables give the best result in predicting code smells?

A DL model's independent variables, or features or predictors, are crucial for accurate predictions. Each metric plays a unique part in determining the code smells. It is categorized according to their design goals to properly analyze the different predictor types working in the primary paper. To list the independent variables used in preliminary studies, this study precisely adhered to the product metrics classification framework. For instance, the LCOM metric was categorized as a Cohesion metric, while the LOC, which measures the length of a source code file, was assigned to the Size category. Each primary study underwent this procedure.

While analyzing papers, it was noted that detecting a code smell affects the number of independent variables. Researchers developed formulae and methodologies employing an independent variable to determine whether a specific code smell exists. For instance, if LOC is greater than 48, then the source code contains a Large Class. Thus, it can be claimed that the independent variable depends on the sort of smell being identified; decisions are made based on the type and number of required metrics.

Table 3.7 describes the metrics condition to identify the existence of a particular code smell. Their meanings are defined in Section 2.4.1.1. Lanza and Marinescu, in their book [167], gave a theoretical overview of the range of different metrics required for code smell detection. Values for this range are determined as *shallow = 1; few = 2-5; short memory cap = 7-8; and many with value > short memory cap*. Still, some conditions depend upon the size and complexity of the dataset particular to each paper. For such situations, there are some typical values that the dataset can define. To determine the values of average high and

low, two statistical methods are employed:

1. **Average (AVG)**: To identify the central tendency or most usual value of the data collection.

2. **Standard deviation (STDEV)**: To determine how widely distributed the values in the data collection are.

After knowing the above values, the following keywords can be given the range as follows:

- Low = AVG − STDEV.
- High = AVG + STDEV.
- Very high = (AVG+STDEV)*1.5.

These margins now clear the meaning of *low*, *high*, and *very high* in Table 3.7.

Most code smell-detecting models developed primarily depend on source code metrics that could detect the structural characteristics of a part of relevant code. Amongst all metrics, the Chidamber and Kemerer metric suite and Lines of Code (LOC) metrics are the most popular and utilized in most primary research. Other valuable metrics included WMC, CBO, NOM, DIT, NOC, and LCOM. This test suite was chosen primarily because it contains metrics that can measure various aspects of source code, such as cohesion, coupling, or complexity. They thus served as the foundation for developing DL methods for detecting code smells.

Numerous additional structural-based measures were also utilized in addition to the metrics used to supplement the evaluation made feasible by the CK suite. The primary research altered specific metrics to represent the structure of source code components more accurately. They specifically build alternatives, such as CK metrics, that do not consider the accessor methods like getters and setters. Instead, other metrics, such as those belonging to the encapsulation and inheritance categories, were considered predictors. This likely demonstrates the research community's openness to investigate novel metrics for enhancing the predictive capability of DL-based code smell detectors.

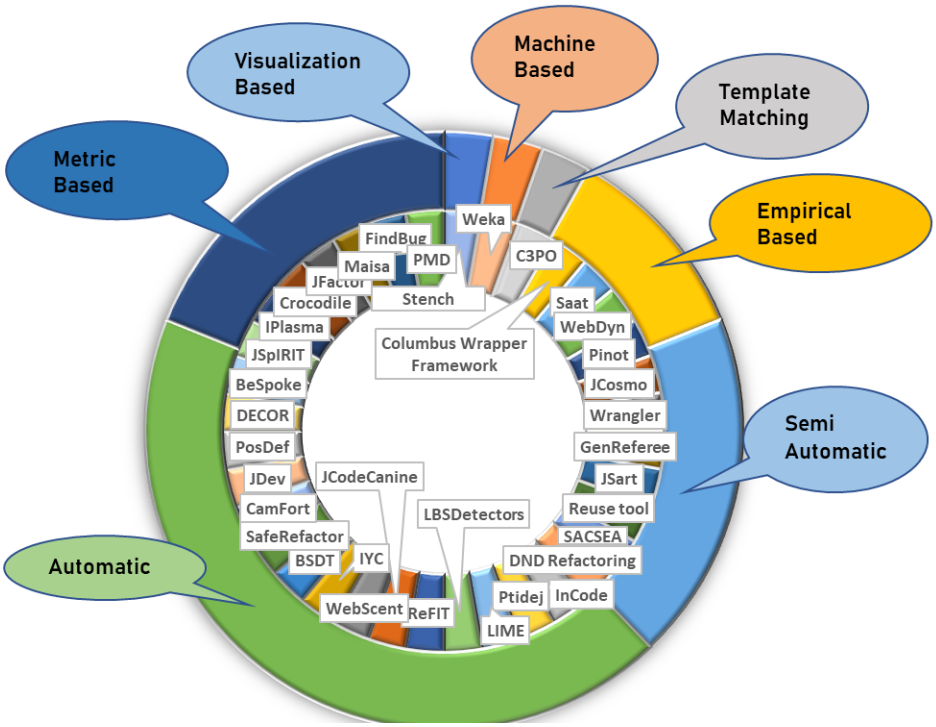


Figure 3.8: Available Tools with their Detection Mechanism

Table 3.7: Logical Representation of Code Smells using Metrics

Code Smell	Metrics	Metrics Category	References
Blob	LOC >high WMC <high TCC «low	Size Complexity Cohesion	[149]
Brain Method	LOC >high/2 CYCLO ≥ high MAXNESTING ≥ few NOAV >many	Size Complexity Complexity Complexity	[46, 150]
Brain Class	Brain Method ≥ 1 LCOM «low CYCLO >few	- Cohesion Complexity	[46, 150]
Complex Class	CYCLO >high DIT >high NOM >high	Complexity Inheritance Complexity	[151],[168],[45],[152]
Complex Method	CYCLO >high LOC >high	Complexity Size	[45]

Table 3.7: Logical Representation of Code Smells using Metrics (Contd.)

Code Smell	Metrics	Metrics Category	References
Data Class	LCOM >low	Cohesion	[168],[50],[169]
	NOAM >high	Complexity	
Dispersed Coupling	MAXNESTING >shallow	Complexity	No detection in paper available
	CINT >short memory cap	Others	
	CDISP $\geq$ half	Others	
Feature Envy	ATFD >few	Coupling	[154],[155],[156],[168],[50]
	LAA <one third	Encapsulation	[153],[169],[138],[170],[157]
	FDP $\leq$ few	Complexity	
God Class	ATFD >few	Coupling	[144],[168],[50],[153],[169]
	WMC $\geq$ very high	Complexity	
	TCC <one third	Cohesion	
Intensive Coupling	MAXNESTING >shallow	Complexity	No detection in paper available
	CINT >few	Others	
	CDISP <quarter	Others	
Large Class	LOC >high	Size	[160],[171],[157],[161]
	DIT >low	Inheritance	
	CBO >high	Coupling	
Lazy Class	LOC >high	Size	No detection in paper available
	NOM $\leq$ low	Complexity	
Long Method	LOC >high	Size	No detection in paper available
	CYCLO «low	Complexity	
Long Parameter List	PAR >low	Others	[172],[45],[152]
Misplaced Class	LCOM <low	Cohesion	[160],[157]
	CBO >high	Coupling	
Multifaceted Abstraction	NOM >high	Complexity	[151],[152]
	DIT >high	Inheritance	
Refused Parent Bequest	BUR <one third	Others	No detection in paper available
	NOP >low	Others	

Table 3.7: Logical Representation of Code Smells using Metrics (Contd.)

Code Smell	Metrics	Metrics Category	References
Shotgun Surgery	CM >short memory cap	Coupling	No detection in paper available
	CYCLO >many	Complexity	
Significant Duplication	SEC >average	Others	No detection in paper available
	SDC $\geq$ 2X(few+1)+1	Others	
	LB $\leq$ few	Others	
Spaghetti Code	CYCLO >high	Complexity	[168]
	LOC >high	Size	
Speculative Generality	NOM >high	Complexity	[168]
Switch Statement	NOC >low	Inheritance	[172]
Temporary Field	TCC >few	Cohesion	[171]
	LCOM >low	Cohesion	
Tradition Breaker	AMW >average	Size	No detection in paper available
	WMC $\geq$ very high/2	Complexity	
	NOM >high/2	Complexity	

To broaden the discussion, the papers reviewed show that only some studies have looked at metrics other than structural ones, which is necessary for antipattern detection. Zhang et al. [46] worked on semantic data analysis. Here, it can be observed that the usage of semantic data expands the scope of correctly detecting code smells. Because these metrics strongly complement one another, DL approaches would perform significantly better if they were more widely used.

None of the studied papers used alternative metrics, such as dynamic or metrics that could distinguish between different concerns. Such metrics fit the detection of many kinds of code smells perfectly. Dynamic metrics, for example, could be effectively used to identify instances of *Message Chains*, defined as methods that perform multiple methods calls. Concurrently, due to their ability to quantify concern's scattering and tangling properties, concern separation metrics are already thought to detect code smells effectively like

Divergent Change and *God Class*. Adding those metrics can significantly improve the effectiveness of current approaches and, more importantly, can be used to detect more code smells at various granularity levels. Many researchers have produced labeled data sets with several independent variables & labeling tools on GitHub.

The more significant point is that none of the earlier studies considered developer-related data (such as workload or experience) that has been successfully used in other software engineering contexts, such as bug or change prediction. Given the subjectivity with which developers interpret these factors in code smell detection, these factors might even be more crucial. In other words, developers can be recommended to beware of code smells that are most important for them at a given time based on their expertise or workload. Therefore, their incorporation offers researchers a significant chance to close the knowledge gap between the field and practice.

Summary of RQ 2.1. The majority of primary studies examined used product metrics. Also, studies went with the combination of different kinds of metrics for efficiency. However, it was found that only a few studies examined the utility of process and text metrics together as an alternative way of identifying code smells. Likewise, it was observed that analyzing semantic metrics improved the effectiveness of code smell detection.

3.3.2.2 RQ 2.2 - Which dependent variable is used to get the results and analysis of code smells?

The dependent variables of code smell-detecting models give information about the severity of smelliness. A classification type defines the sort of output that a code smell prediction model generates. Binary categories and probability ranges are the most common ways of determining dependent variables. In the first situation, a model seeks to determine whether or not a smell afflicts a software artifact. In contrast, the latter aims to forecast the possibility of a source code entity being smelly, typically ranging between 0 and 1. The fundamental distinction between the two classification approaches is the choice of a threshold to separate those objects. From this study, it is observed that 60% of the papers used binary classification while 40% went for probability classification.

There are mainly two reasons why binary type classification is so popular: firstly, it makes the dataset categorical, and secondly, predictions in DL models become easier. Numerous code smells in a single dataset become readily identifiable. This SLR also contains datasets in which a boolean type dependent variable is employed to determine whether or not a specific code smell is present in the dataset by giving the output as TRUE/FALSE.

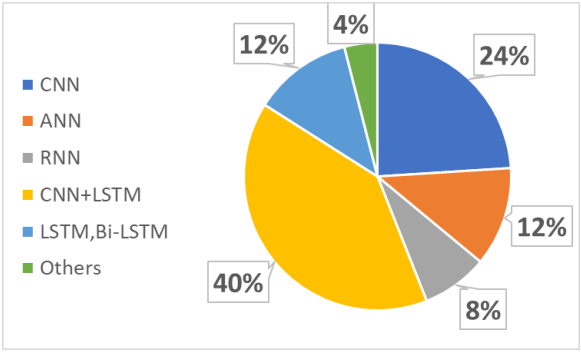


Figure 3.9: Commonly used Deep Learning Algorithms

Studies that defined classification methods had two types of results. Either the code is (i) smelly or (ii) non-smelly. Code smells like *Temporary Field, Large Class, and Misplaced Class* had their smelliness predicted using probability classification wherein the percentage of the range of code smell was given in the project. Other code smells were identified using binary classification, which could tell if the program was smelly. Four code smells have been tested across both classification types: *God Class, Long Method, Feature Envy, and Data Class*. It can be seen that both categorization types are effective from their respective point of view because the results from both are precise and clear.

Summary for RQ 2.2. Two classification categories are identified: binary classification and probability classification. Through investigation, it was found that 40% of primary studies worked using probability and 60% using binary classification type. According to the findings, both have worked satisfactorily, and newer approaches can be formed.

3.3.2.3 RQ 2.3 - Which DL algorithms are utilized in code smell detection?

The algorithm of each model is the most crucial component. This review featured various articles that used some of the most common DL algorithms, including CNNs and ANNs. This result is summarised in Figure 3.9. Diversifying the number of CNN layers improved the prediction accuracy. While some researchers found that increasing the layer count boosted prediction accuracy, other studies found that changing the layer count made no impact. Consequently, it can be seen that the precision is independent of the quantity of CNN layers. While using any DL model, the dataset must be augmented. Since there were limited datasets with pre-augmentation, methods required a data augmentation technique that could address the issue of imbalanced datasets. Through investigation, it was found that the Synthetic Minority Oversampling Technique (SMOTE) is one of the most popular data augmentation techniques. It is a statistical approach that enables the

model to increase the number of instances in the data set in a balanced manner. The component creates new illustrations from existing minority situations specified as input.

Continuing with the analysis, some papers advanced the level of model creation by combining two or more learning layers to create a more efficient layer. The model was also fitted with additional DL neural networks, including RNN, LSTM, and Bi-LSTM, extensively employed with CNN. When combined with CNN layers, they performed more accurately. In most situations, combining these models worked better than a single-layer model. This shows us that the best features of more than one layer, combined with the basic CNN layer, can vary and even increase the chances of getting above-fair results.

Dense layers and output layers were finally used to finish the models. The number of thick layers varied from paper to paper, but the real impact was caused by the specific DL algorithm used in previous layers. Hence, it was observed that there is no constraint on the number of dense and output layers to the effectiveness of the models against antipattern detection.

Summary for RQ 2.3. It is observed that CNN and RNN are the most used algorithms with a combination of different DL layers, including LSTM and Bi-LSTM, strengthening the models' precision. Amongst all, CNN and LSTM layers are the most effective DL algorithms, as 29% of the studies used them.

3.3.2.4 RQ 2.4 - What detecting tools are most commonly used in research for code smells?

Researchers have employed detection tools in papers for code smell identification and performing possible refactoring. Figure 3.8 presents the list of tools utilized in detection. Some of the most essential and frequently used tools include jDeodorant and iPlasma, which are metric-based detection tools. Both of these tools are Eclipse plug-ins. An Eclipse plug-in is an extensible framework applied to individual software that locates and fixes software design issues by implementing the proper refactoring. JDeodorant uses several cutting-edge approaches and methods to spot code smells and provide the right refactoring to fix them. The tool currently recognizes five categories of bad code smells: *Feature Envy, Type Checking, Long Method, God Class, and Duplicated Code*. Other utilities, besides the Eclipse plugin, are available as independent software. They can function like an Eclipse plugin without IDE and require a complete setup before being utilized.

Some automatic detection tools, such as BDST and jDev, are available for refractive imaging. Then, a semi-automatic detection technique called SACSEA is included in the list. Despite being semi-automatic, it is utilized to reflect the most prevalent code smells, such as *God Class and Feature Envy* and not-so-popular smells like *Magic Numbers and Global variables*. Other popular tools include JProfile, NIPAD, PMD, infusion, DECOR,

refactoring assistance, and CCFinder.

Summary for RQ 2.4. The research methods make excellent use of detection tools. It is observed that metric-based tools are used most commonly. Apart from this, both semi-automatic and fully automatic tools are often employed. Both were found to be equally valuable for detecting standard code smells.

3.3.2.5 RQ 2.5 - What strategies have been adopted for training the models?

DL models are trained using two forms of training data: within and cross-project. Training a model using data from the same project or dataset for which the model will be used is called "within-project training." Such training methods are utilized when a model is being constructed for a particular job or issue, and the training data also represents the actual data well. The benefit of within-project training is that it enables the model to become familiar with the particular patterns and characteristics of the data used, which may improve its performance on that data.

Contrarily, cross-project training entails building a model using information from a different project or dataset than the one it will be applied to. This training is frequently utilized when the target project lacks labeled data or the model has to be trained on more varied data to enhance its generalization capabilities. Cross-project training can help the model discover more universal patterns and traits that hold across several datasets, which could improve performance on new, unexplored data.

It is important to remember that cross-project training might be more complex than within-project training since the data may have different properties or distributions from the target dataset, which can result in over-fitting or subpar generalization performance. It is crucial to carefully choose the cross-project data and utilize the proper methodologies, such as data augmentation or transfer learning, to aid the model's adaptation to the new data.

Most algorithms of prediction models developed so far have had their effectiveness assessed in intra- and inter-project contexts, resulting in the training of the same model twice, once using data from the project under study and once taking help from other project datasets. The training strategy is enhanced when a project is applied to cross-project and within-project datasets. If a given approach is evaluated across projects, it should also be employed within projects to benchmark its genuine capabilities fairly. Hence, researchers should focus more on cross-project training for efficient results.

Summary for RQ 2.5. While 40% of the code smell predicting models were only considered in a cross-project situation, most were tested in both cross- and within-project settings. It was found that cross-project training gave better results.

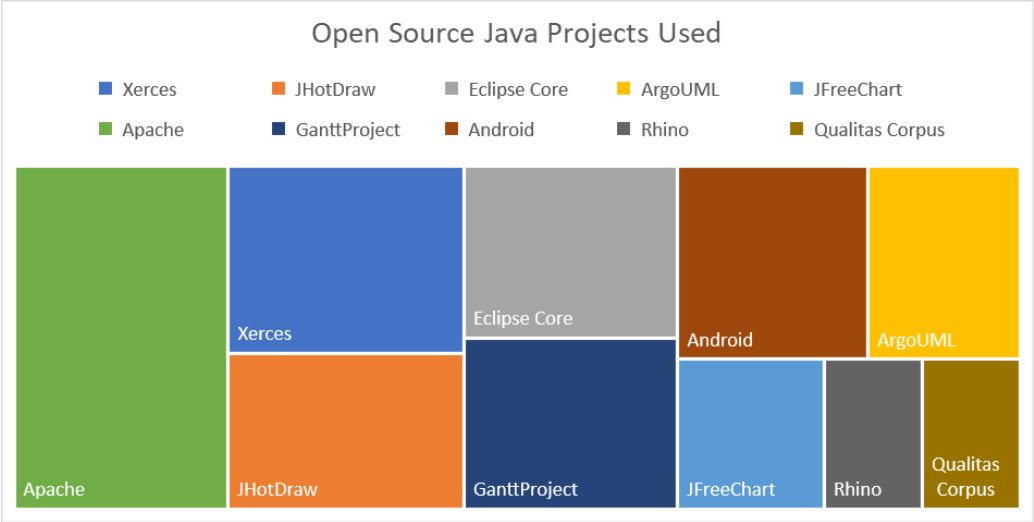


Figure 3.10: Commonly used Datasets in Detecting Code Smells

3.3.3 Results Specific to RQ3

The third research question focused on the assessment methods for the code smell predicting model. It includes (i) datasets exploited in this domain and (ii) outcome metrics utilized for result summarization.

3.3.3.1 RQ 3.1 - What datasets are used to detect code smells?

The analysis of the datasets used in the publications is of utmost importance for comprehending the assessment setup of the primary research. To analyze papers efficiently, the authors agreed to review documents written in a particular language. Object-oriented language efforts, particularly those in Java, were the focus of the research papers and journals. This is because most code smells were discovered in Java programs, one of the most widely used object-oriented languages. However, the primary obstacle to continuing with Java is that most Java projects are private. Due to this, numerous researchers searched GitHub for various industrial and publicly accessible Java scripts. The primary source of publicly available datasets to these researchers is shown in Figure 3.10. As can be seen, Apache provides the complete sets of data in Java. Xerces, JHotDraw, Eclipse, Android API, and others comprise the top part of the dataset sources. These codes were formally implemented and examined for any code smells using automatic and manual detection.

Some researchers went through both forms of detection to increase their precision. To double-check the accuracy of the computational check, manual detection was performed after the data set had been submitted for automated detection. This required significant

time and money. Some researchers created private datasets for their research. This was generally done to create a dataset with unique variables specifically used for detecting a particular code smell the research focuses on. Another condition where datasets were manually created was when there was a shortage of codes containing code smells. Since this is the latest study, codes consisting of code smells are limited, and researchers need a large amount of data heuristically formulated for model training. In such situations, refactoring was applied to generate datasets according to the requirement.

Summary for RQ 3.1. This section concluded by providing the primary sources for openly available datasets to researchers working on this study. Apache is the most widely used publicly available dataset. Due to a shortage of code smell-containing datasets, few researchers generated datasets using refactoring. Publicly available datasets are open for reproduction, but manually generated datasets are private.

3.3.3.2 RQ 3.2 - Which result metric is used to evaluate the models?

Figure 3.11 shows all the different evaluation metrics used by primary studies. The most often used statistic by almost ten studies has been *F1-score*; *Accuracy* score was available for 13 papers. Some papers described their result section with a combination of three metrics mainly- F1 score, precision, and recall. The results of the three metrics complement each other and give more significant details of the result metrics.

Positive rate metric provided information about the proportion of detected code smells in the article. Authors of papers Gupta and Singh [171] and Palomba et al. [173] used this method to detect the percentage of code infected by smells. Other least frequent metrics included the MCC. Yu et al. [169] compared the MCC values of different algorithms applied and formulated the results. *MCC* is a reliable metric summarising the classifier's performance to one value, assuming positive and negative examples are given equal weight.

The *AUC*, a metric commonly employed in binary classification, has emerged as a prominent tool for assessing the outcomes of code smell detection. This measure was particularly prevalent in studies like [165] showcasing a range of values from 87.4% to 95.8%. Likewise, [162] leveraged AUC to achieve values around 95%. The utility of AUC curves lies in their capacity to illustrate diversity through the alterations in curve patterns. The graphical representation indicates a prevalent preference for accuracy as the primary metric employed in code smell detection research. Additionally, it illustrates consistently elevated outcomes within this metric, accompanied by reduced loss values.

Although threshold-dependent measures like F1-score, accuracy, and recall are frequently utilized, they can occasionally be biased in how well the models perform. Con-

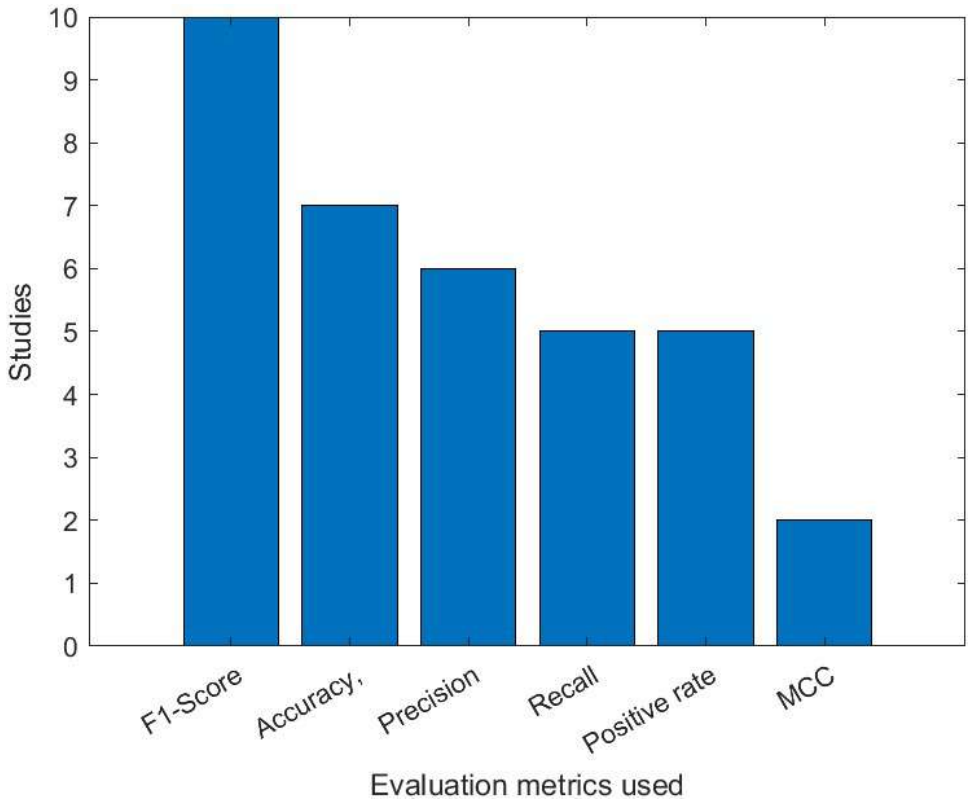


Figure 3.11: Evaluation Metrics Used for Result in Primary Studies

sequently, if they are supplemented with threshold-independent measures like MCC, the findings will be more effective.

Summary for RQ 3.2. The most commonly used metric for result evaluation is Accuracy, with 13 papers using it, followed by F1-Score, Recall, and Precision. Using threshold-independent metrics like MCC together with dependent metrics can give comparably better results.

3.3.4 Results Specific to RQ4

The efficiency in code smell predicting models is subject to a statistical meta-analysis presented in this final research question. For such investigations, the prediction models should be re-run on a common dataset with a similar evaluation standard to be easily compared. However, SLRs are of limited scope, so the goal is to synthesize the results from significant studies while considering the limitations of the previous studies. In this work, the focus is on how (i) independent variables, (ii) DL methodology, and (iii) training technique influenced the performance shown by these code smell prediction models.

Nearly 90% of the publications employed F-measure, accuracy, and recall as the main result analysis metrics for the performance calculation. This made it easier for us to eval-

Table 3.8: Evaluation metrics reported in primary studies

ID	Ref	F1-Score	Recall	Precision	Accuracy
S1	[43]	83	NR	NR	NR
S2	[166]	91.7	92.3	92	NR
S5	[174]	NR	NR	NR	97.58
S6	[175]	NR	NR	NR	75
S7	[50]	NR	NR	NR	95.75
S8	[153]	NR	NR	NR	99
S9	[46]	NR	NR	NR	4.41*
S10	[155]	43	35	40.8	1.02*
S12	[45]	74	65	86	NR
S13	[176]	NR	NR	NR	87
S14	[161]	NR	NR	NR	94.21
S17	[149]	NR	NR	NR	96.84
S18	[163]	NR	96	78	NR
S19	[177]	98	97	98	NR
S20	[49]	NR	NR	NR	2.01*
S21	[178]	22	35	16	NR
S22	[157]	NR	NR	70	80
S23	[156]	55.97	85.35	47.09	NR
S25	[179]	77	86	71	NR
S27	[150]	NR	NR	NR	97
S32	[180]	22	16	99	NR

uate the performance of various publications. This section will discuss how the three factors mentioned above have impacted each paper's results section. Table 3.8 describes the studies that used the above criteria with their results. "NR" denotes the absence of a reported metric for a particular study. Studies indicating metric improvements compared to prior algorithms without providing the precise values for their proposed method are marked with an asterisk "*" symbol.

In contrast to the inverse variance approach, this research paper has given precedence to all the chosen publications, irrespective of the dataset size or the research domain. Identifying optimal combinations of algorithms and tools for achieving the most favorable results necessitates a thorough analytical investigation.

3.3.4.1 RQ 4.1 - How do independent variables impact performance?

According to the study, numerous independent factors are needed to identify code smells. While some studies consider structural elements, others also think of semantics. However, the argument remains that the more independent variables complicate the process,

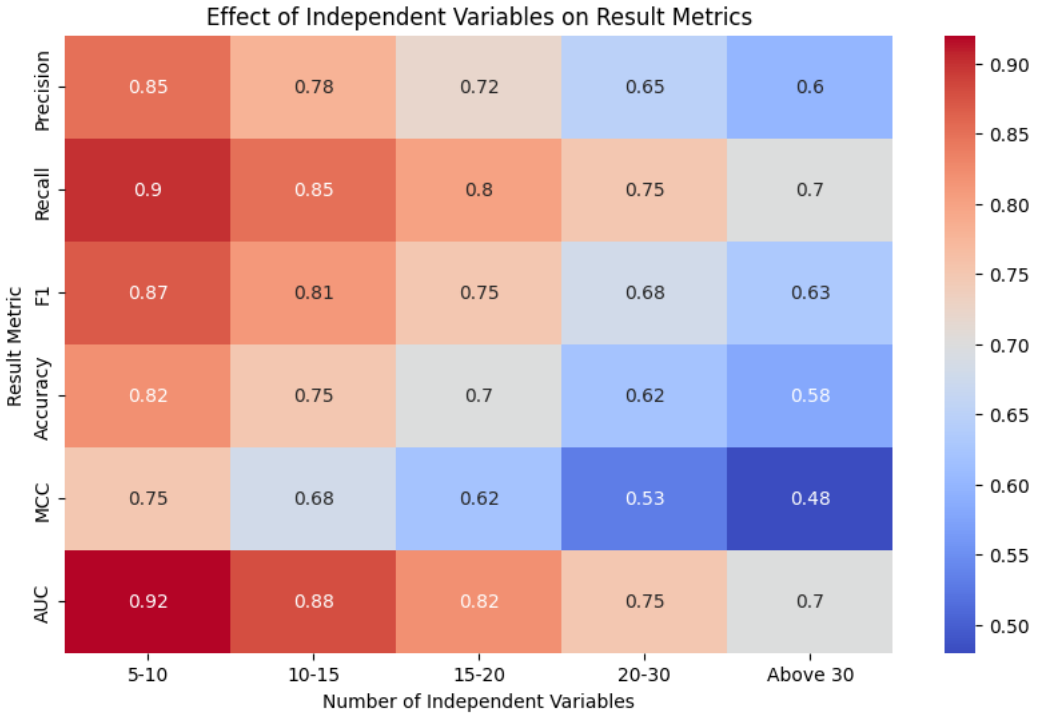


Figure 3.12: Performance analysis for Independent Variables

so more time is required. As a result of this, the model may end up being over-fitted. The papers also considered several intricate variable columns. Therefore, the impact it has reduces the accuracy of internal project models as well as cross-fitting models.

The heatmap depicted in Figure 3.12 illustrates the influence of the number of independent variables on the performance metrics. Notably, reducing the usage of variables to focus solely on the essential ones for code smell detection leads to a heightened emphasis on critical features, consequently yielding improved outcomes. Conversely, incorporating numerous independent variables boosts the risk of diverting attention towards less pertinent features, potentially causing the model to overfit in certain instances. Among the performance metrics, the most substantial variation arising from alterations in the variable count is observed in the AUC, indicating its sensitivity to selecting appropriate variables.

Summary for RQ 4.1. Reducing the count of independent variables is imperative, as it directly impacts the model's accuracy of code smell detection. Among the various performance metrics, the one most profoundly influenced by adjustments in the independent variable count is the AUC.

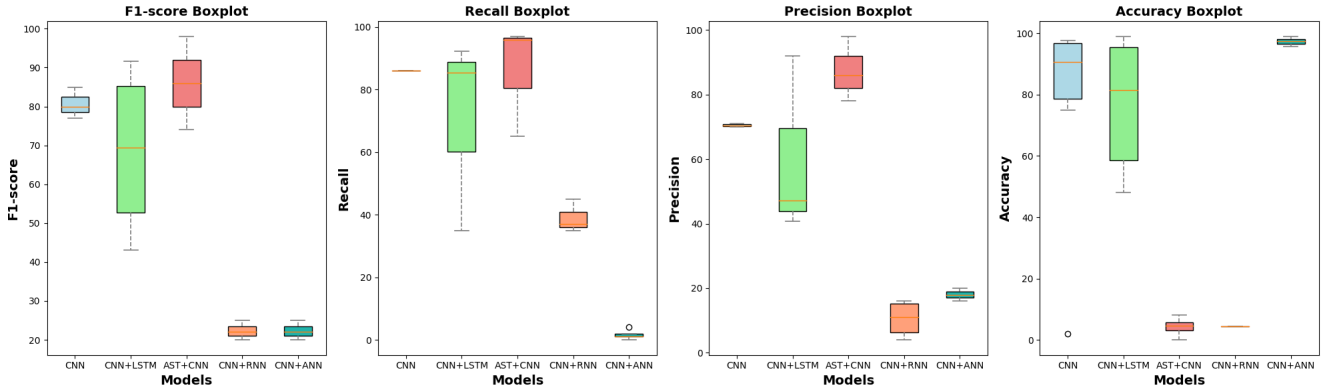


Figure 3.13: Performance metrics for different DL models

3.3.4.2 RQ 4.2 - Is the performance of code smell prediction models affected by DL algorithms?

The DL algorithms used in the papers showed that convolution neural network (CNN) was the most used algorithm. Only using convolution layers increased precision up to an average of 90%, but it was investigated that increasing convolution layers in the model had little or no improvement in performance. The results in Figure 3.13 present a comparative analysis of various DL models and their combinations based on different result metrics. Notably, the combination of CNN and LSTM consistently demonstrates the most favorable outcome across all four performance metrics. These metrics always exhibit values that fall within the higher range, indicating the attainment of remarkable results. The depicted boxplot is a valuable tool for assessing and contrasting the effectiveness of distinct deep-learning models about diverse result metrics.

The performance models are improved by adding more DL layers to the convolution layers. LSTM layers included in the CNN layers increase the results in precision- 0.0408, recall - 0.0035, and f1 - 0.043 times. In another paper, Bi-LSTM layers increased the development as precision- 0.83, recall- 0.74, and f1- 0.78 times. One of the papers used CNN and GRU together. This enhanced model increased the model accuracy by 4.41%.

Summary for RQ 4.2. It is observed that using different types of DL layers increased the accuracy. Using algorithms like LSTM, Bi-LSTM, and GRU gives efficient results.

3.3.4.3 RQ 4.3 - What is the impact of Training Strategies

As discussed earlier, the models are trained by two means, either within the project via splitting the dataset in training and testing or by cross-project. Within the project, splitting gave a way to create the algorithm using the dataset in the same format. There was no

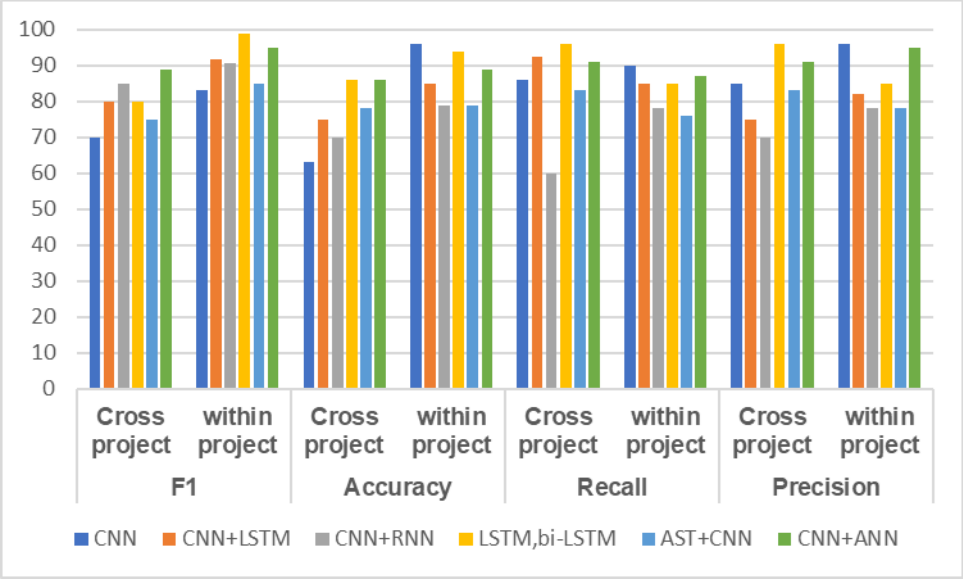


Figure 3.14: Performance metrics for different Training Strategies

need to pre-process or make the columns similar using tools. This enhances the efficiency of the model as well. Data is first brought to a similar format as the training data in cross-project testing. However, this section analyzes them equally and fairly based on their Odds Ratio (OR). Accordingly, checking the accuracy given by both types of strategies shows that the Odds Ratio (OR) of within-project models is better than cross-project ones.

Figure 3.14 visually presents the performance of distinct models about cross-project and within-project strategies, aiming to deliver outcome metrics. In most instances, it becomes evident that within-project training strategies yield more favorable results.

Summary for RQ 4.3. Variation in data contributes to the prediction of code smells. It is observed that within-project models perform up to 19% better than cross-project ones. Therefore, it can be said that within-project models increase the efficiency of the models.

3.4 Discussions

DL for code smell detection is a very young and expanding study area. As a result, most of the papers have employed DL techniques in their papers and identified research gaps that require more attention. Some studies emphasized these weaknesses and even indicated the areas needing repair. Future work was a significant subject of these papers. In other research, future work signified their next move towards more innovation. This section discusses the research gaps present in the primary studies and possible plans for the future.

3.4.1 Outcomes of Literature Review

RQ 1. - *Limited number of code smells detected using DL approaches due to limited resources:* From this research question, it can be formulated that a minimal number of code smells like *Feature Envy, God Class, Long Method, and Complex Class* have been detected using DL algorithms. However, most of the other smells given by Fowler, Brown, and Wake, like *Spaghetti Code, Refused Bequest and Functional Decomposition* mentioned in Table 2.1 and 3.1 have not yet received much attention. Hence, it becomes the most typical area for development to broaden the research to find other code smells, too. Due to the small number of projects and the limitations of available resources and technological advancements, just a tiny percentage of code smells could be detected in a single thesis. A collection of code smells was found comparable to specific publications to be discovered simultaneously. This made it possible to improve the algorithm to meet future requirements, such as finding *Brain Class and Brain Method* simultaneously. So, efforts should be made to detect multiple code smells simultaneously to improve efficiency. Code smells, which are more frequent, may also be detected using the same method to find them.

RQ 2. - *Lack of information about metrics selected for detecting particular smell:* Selecting independent and dependent variables is one of the key methodology components of research. The papers discussed how several independent variables are required for continuing studies to identify a certain code smell. However, no paper gave concrete information about the metric rules that have been selected to detect the smell. Hence, it becomes difficult for the researchers to identify the correct set of independent variables. The basic metric rules and their selection strategy for some code smells have been given in Table 3.7, which can be used for strategy formulation. It is also observed that the detection strategy for some smells largely depends on the size of the dataset. Hence, a future project will also improve feature selection on such datasets for efficient time management.

RQ 2. - *Hybrid algorithms for detection and optimization:* DL method is one of the most often studied future directions. After completing their research, several academics realized that expanding the combinations of many DL layers might be a more practical approach toward code smell detection. A few researchers put out the notion of adding LSTM layers to the CNN model. One of the papers recommended utilizing Natural Language Processing (NLP) more effectively for better results. When the area of DL is acquainted with the field of code smell, researchers also want to extend this notion to **quantum computing**. The problem of code smell also exists in quantum computer-written

codes. Development for expanding the idea of detecting code smell in quantum computing is still ongoing since the design and syntax of these codes differ from standard OOP languages. It is expected that the number of quantum papers for validity could have been higher than expected. **Use of genetic algorithms to optimize the performance of the models is also one of the future directions for researchers.** Utilizing genetic algorithms is essential for devising an optimized approach for metric parameters, thereby enhancing both model training efficiency and the detection process. Research into optimized techniques and evolutionary detection within the research domain holds potential.

RQ 3. - *Limited number of openly available datasets:* Due to the increased source and code formulation, it takes longer to create a great dataset. A minimal set of openly available datasets can be used for code smell detection. Existing datasets have been either exhausted to detect code smells or possess only some code smells. **Area of *Generative AI* can be explored to manually generate new structural and semantic datasets in a time-bound manner.**

RQ 4. - *Meta Analysis and Language restrictions :* The meta-analysis emphasized that various factors, namely the independent variables, DL algorithm, and training strategy, can significantly influence the performance of DL approaches. This finding is crucial for future research, as it underscores the importance of thoroughly evaluating the impact of each aspect related to the settings of DL approaches. Doing so makes it easier to interpret the performance and understand how the technique responds to different environments. Additionally, primary studies mainly concentrated on Java code but can be extended to other object-oriented languages, such as Python, C, C++, etc. Language extension should also be employed to widen the scope of available semi-automatic and automatic detection tools.

3.4.2 Research Gaps

This section identifies open issues or research gaps in the primary studies considered in this SLR. It is observed that DL algorithms could identify a particular group of code smells. The number of code smells detected can vary by increasing the independent variables or introducing new detection tools. This issue can be a focus of concern for further research advancements. Relying on a single benchmark dataset and a small number of code smells for experimentation cannot produce appropriate results. This can lead to negligence in adjusting model hyperparameters before training to reach a local optimum, affecting overall performance. Another study discusses their drawback as relying on a single benchmark dataset and a small number of code smells for experimentation. They

also neglected to adjust the model hyperparameters before training, which led to a local optimum and impacted overall performance.

As mentioned in section 3.3.2.4, SMOTE is the most used data augmenting technique. It is helpful in this field, but a few papers showed no improvement in detecting God class code smell using SMOTE. Hence, a new approach other than SMOTE should be brought to use. Another area for improvement is that more optimization points lead to subpar performance. It demonstrates both the need for optimization points and the viability of their concept. Here, evolutionary algorithms can be used to achieve such optimization.

Apart from them, issues like exploring the field of quantum computing and genetic algorithms are very critical. As discussed, quantum code smells are evolving and can threaten software maintenance in quantum computing. Chen et al. in [70] briefly explained some code smells seen exquisitely in the quantum field. It requires early detection and removal tools. However, research in this field is rare as of now. Similarly, there are steps where researchers are trying to accomplish detection by genetic algorithm or nature-inspired algorithm to detect multiple smells simultaneously.

Chapter 4

DIR-SMOTE: A Density-Influence Resampling Framework for Imbalanced Code Smell Detection

This chapter tackles the challenge of class imbalance in code smell detection, where smells like Data Class and Feature Envy are often overlooked. To address this, it introduces **DIR-SMOTE**, a resampling method that combines density filtering with SHAP-based feature importance to create meaningful synthetic samples. Evaluated on five benchmark datasets, DIR-SMOTE consistently outperforms existing techniques, improving precision and F1-score while remaining interpretable and efficient. The remainder of the chapter is organized as follows: Section 4.1 describes the introduction section, Section 4.2 outlines the proposed methodology employed in this study. Section 4.3 details the proposed DIR-SMOTE algorithm for imbalanced data. Section 4.4 discusses the results, and Section 4.6 presents the discussions. The work presented in this chapter has been published in Automated Software Engineering Journal [181].

4.1 Introduction

Code smells are indicators of potential issues in software design that can lead to maintenance challenges and decreased code quality [182, 183]. These structural defects, such as duplicated code or excessive complexity, can impede the development process and impact the long-term sustainability of software systems. Addressing code smells is crucial, as their presence can make code error-prone and challenging to maintain [169]. Consequently, code smell detection has become a significant area of research. Previous studies have explored various detection processes but faced common issues, including the

need for extensive datasets for training and evaluation. Most research has been limited to languages like Java and Python [184, 185], often resulting in smaller datasets and less promising results. Additionally, available datasets frequently suffer from imbalance, with a surplus of non-smelly data compared to smelly samples. This imbalance reduces training models' precision, recall, and F1-score, potentially leading to biased outcomes [186].

To mitigate these challenges, sampling techniques can be employed to balance the dataset [187]. Oversampling increases the representation of the minority class using methods such as SMOTE [188, 189] or ADASYN [190, 191], which generate synthetic data points. While this improves model performance and reduces bias, it can also lead to overfitting if the synthetic data too closely resembles existing samples. Conversely, undersampling minimises the size of the majority class, employing techniques like Random Undersampling (RUS) [192], Edited Nearest Neighbours (ENN), or Tomek Links (TL) [193] to either randomly discard examples or refine class boundaries. Although undersampling can lower computational complexity and training time, it risks losing valuable information from the majority class, which can impact model generalisation [194, 195].

Hybrid sampling methods combine the strengths of both oversampling and undersampling. Techniques like SMOTE-ENN and SMOTE-TL [196] apply oversampling to the minority class while also cleaning the dataset to remove noise and overlapping instances from the majority class. This approach aims to improve class representation and decision boundary clarity, balancing the benefits of oversampling and undersampling while mitigating their respective drawbacks. However, these traditional methods suffer from several drawbacks. They often generate synthetic instances blindly, without considering feature-level importance or sample density. Moreover, these methods lack semantic awareness, which leads to poor-quality synthetic samples. Additionally, the interpretability of these techniques remains limited, making it difficult to understand their impact on model learning [197].

To address these limitations, this study introduces a novel resampling technique called Density and Influence-based Resampling using SMOTE (DIR-SMOTE). Unlike conventional methods, DIR-SMOTE incorporates two major innovations: density-aware oversampling and SHAP based feature influence. First, the method uses density computation to identify sparse regions of the minority class and selectively generates synthetic samples in these low-density areas, ensuring better coverage of the decision space. Second, DIR-SMOTE integrates SHAP to guide the synthetic sample generation process toward features that significantly influence model predictions. This produces more semantically valid and interpretable synthetic samples [43].

SHAP is employed due to its additive attribution property and its compatibility with tree-based learners commonly used in code smell detection. Although correlated software

metrics may introduce instability in SHAP values, this effect is mitigated through the use of TreeSHAP and by interpreting SHAP values primarily as relative importance indicators rather than causal explanations. When combined with density-based constraints, SHAP-guided perturbation limits excessive variation across correlated metrics and helps preserve meaningful metric relationships during sample generation [43]. In addition, SHAP is used after model training to interpret feature relevance and validate the semantic consistency of the resampled data

By combining density estimation and XAI in the form of the SHAP method, DIR-SMOTE offers a balanced, interpretable, and high-quality resampling strategy that enhances minority-class learning and mitigates overfitting [42, 44].

To validate DIR-SMOTE, the study uses five benchmark datasets spanning binary and multiclass smell classification: Apache, jEdit, EDTForCSD [198], DesigniteJava, and MLCQ. These datasets provide a comprehensive testbed across open-source and industrial codebases. A wide spectrum of resampling strategies is compared, including baseline (no resampling), oversampling (SMOTE, ADASYN), undersampling (TL, ENN, RUS), and hybrid methods (SMOTE-TL, SMOTE-ENN) [129, 199].

Additionally, SHAP is used not only during the resampling phase to guide the generation of synthetic data but also after model training to interpret feature importance and validate the semantic alignment of the resampled instances.

This study is driven by the following RQs:

- **RQ1: Which base classifier yields the highest performance for the initial model training across imbalanced datasets?**
  This question aims to identify the most suitable classification algorithm (e.g., DT, RF, XGB) for learning from imbalanced data. Evaluating various classifiers helps ensure the model is robust and provides accurate predictions before applying explainability and resampling techniques.

- **RQ2: How effectively can SHAP-based explanations identify the most influential features for synthetic data generation?**
  This question investigates the role of SHAP values in feature importance ranking. By understanding which features contribute most to model predictions, the study ensures that synthetic samples are generated along the most impactful dimensions, preserving the decision boundary.

- **RQ3: Which density estimation technique provides the most reliable weighting for minority class samples?**
  This question explores the impact of various density estimation approaches in guiding the number of synthetic samples generated per minority instance. The focus is

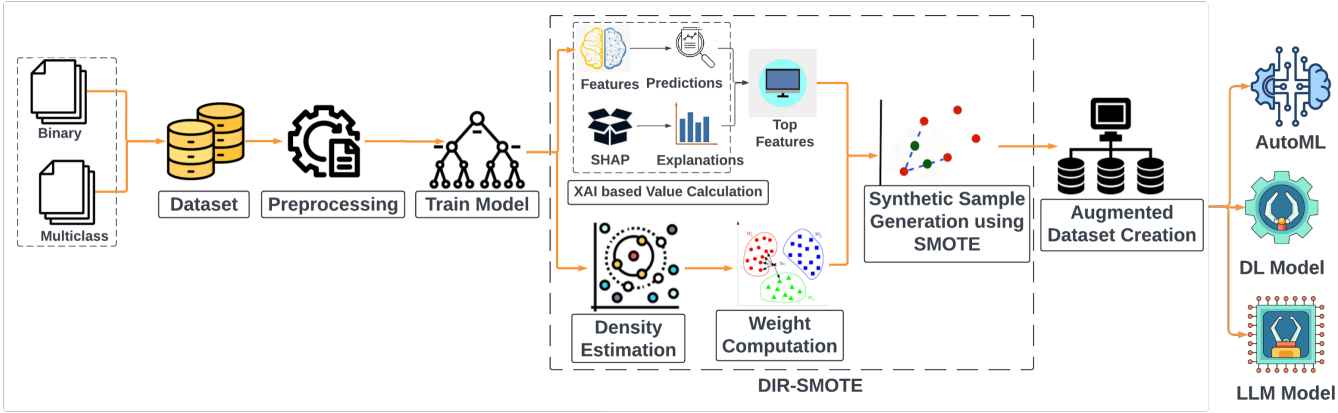


Figure 4.1: Proposed Methodology for Enhancing Code Smell Detection Using Sampling Techniques and Different Learning Paradigms

on using neighbourhood information (like KNN to assign higher weights to isolated or sparse points in the minority class.

- **RQ4: How does DIR-SMOTE perform against existing oversampling methods in improving classification on imbalanced datasets?**
  This question evaluates the effectiveness of the proposed DIR-SMOTE algorithm compared to traditional techniques such as SMOTE, ADASYN, and BL-SMOTE. The comparison focuses on metrics like precision, recall, F1-score, and MCC to validate DIR-SMOTE's performance.

- **RQ5: How does DIR-SMOTE perform across different learning models?**
  This question examines whether the effectiveness of DIR-SMOTE depends on a specific classifier or whether it remains consistent when used with different learning models. For this analysis, the datasets augmented using DIR-SMOTE were evaluated using three different modeling paradigms: AutoML based classifiers, DL models, and LLM-based classifiers.

4.2 Methodology

Figure 4.1 illustrates the step-by-step workflow of the proposed methodology. The pipeline initiates with the collection of five benchmark datasets: jEdit, Apache, DesigniteJava, EDTforCSD, and MLCQ, each exhibiting varying degrees of class imbalance. These datasets are subjected to a uniform preprocessing phase to ensure consistency in format, feature representation, and class labelling.

Table 4.1: Dataset Description with Training and Testing Splits and Smell Category

Dataset	Metrics	Smelly		Non-Smelly		Total	Imbalance Ratio	Smell Types
		Train	Test	Train	Test			
jEdit [122]	34	212	91	1012	434	1749	0.21	Data Class, Long Method, Feature Envy, God Class
Apache [173]	34	3722	1595	6893	2298	14508	0.54	Long Method, Feature Envy, God Class
EDTforCSD [127]	28	586	146	58542	14636	73910	0.01	Spaghetti Code, Data Class, Feature Envy, Refused Bequest
MLCQ [128]	34	1162	498	10258	3082	15000	0.11	Blob, Data Class, Long Method, Feature Envy
DesigniteJava [129]	35	2307	865	19005	5823	28000	0.12	Blob, Large Class, Lazy Class, Refused Bequest

Following preprocessing, the DIR-SMOTE algorithm is applied to mitigate the imbalance by generating synthetic samples for the minority class. This resampling approach incorporates two major innovations: density-awareness and feature influence weighting via SHAP values. The integration of these components ensures more informed and targeted augmentation, particularly in sparse and underrepresented regions of the feature space.

Subsequently, the augmented datasets are passed through a classification stage, where different models are trained to assess the performance impact of the resampling strategy. Finally, the evaluation phase employs multiple metrics to analyse the predictive capability of the models and validate the efficacy of the DIR-SMOTE method.

4.2.1 Datasets Used

To validate the efficacy of the proposed DIR-SMOTE approach, five benchmark datasets, namely, jEdit, Apache, EDTforCSD, MLCQ and DesigniteJava, were employed, representing a diverse range of software systems and code smell scenarios. The details of each dataset is described in Section 2.6.2. Each dataset contains labelled instances of both smelly and non-smelly code components, and the datasets exhibit varying levels of class imbalance, reflecting real-world distribution challenges. A summary of the datasets, including their training and testing splits, total instances, and imbalance ratios, is presented

in Table 4.1.

4.2.2 Feature Representation and Input Construction

The datasets used in this study consist of static software metrics commonly employed in metric-based code smell detection. A core set of object-oriented metrics is available across most datasets, including LOC, WMC, CBO, DIT, NOC, RFC,LCOM, and CYCLO. Some datasets further include additional metrics such as NOM, Fan-in, and Fan-out.

The metrics were extracted using the tools originally employed by the dataset authors:

- **jEdit and Apache** : extracted using the CKJM tool (version 1.9) as described in [122, 173].
- **EDTforCSD** : extracted using the same tool chain as in [127] (a custom metric collector).
- **MLCQ** : extracted using a combination of CKJM and the Eclipse Metrics Plugin [128].
- **DesigniteJava** : extracted using the DesigniteJava tool (version 2.1) [129].

All datasets are publicly accessible at the URLs provided in the respective citations. As the datasets originate from different sources and metric extraction tools, their full feature sets are not identical. Each dataset is processed independently, and classifiers are trained using the complete set of available metrics for that dataset, preserving the original feature space. For each software entity, the input consists of a normalized numerical feature vector that captures design properties such as size, complexity, coupling, and cohesion. This representation follows established practices in code smell detection research and ensures that evaluations are conducted on realistic, unaltered data while maintaining comparability through the shared core metrics.

The selected metrics have direct semantic relevance to the code smells studied. For example:

- **High CBO** (coupling) often indicates excessive coupling, a key symptom of *Feature Envy* and *God Class*.
- **High WMC** and **CC** (complexity) are characteristic of *Long Method* and *Complex Class*.
- **Low LCOM** (cohesion) suggests low cohesion, typical of *God Class* and *Blob*.
- **High LOC** (size) is a common indicator of *Large Class* and *Long Method*.

This semantic alignment is crucial for the proposed DIR-SMOTE framework because SHAP attributions identify which metrics drive the prediction for each instance. By knowing the semantics, it can be interpreted why a feature is marked as influential (e.g., if SHAP gives high importance to CBO for a *Feature Envy* instance, it confirms that coupling is the main discriminative factor). DIR-SMOTE then preserves such semantically important features during synthetic sample generation, ensuring that the generated instances remain realistic and meaningful.

4.2.3 Preprocessing

Before applying the proposed DIR-SMOTE resampling technique and subsequent classification models, a thorough preprocessing pipeline was employed to ensure consistency and enhance the quality of feature representations across all five datasets. All datasets were converted into a unified format to ensure compatibility. Each dataset was parsed to extract relevant code metrics or software features, and entries with missing values or non-numeric attributes were either removed or imputed using median values. Additionally, duplicate entries and irrelevant columns were discarded to retain only those features useful for smell detection.

4.2.3.1 Normalization

The datasets varied in how code smells were labelled. For consistency, all smelly instances were encoded as '1' (positive class) and non-smelly instances as '0' (negative class). In multi-label scenarios (e.g., where multiple smells exist), binary classification was performed on a per-smell basis except for the EDTforCSD dataset on which multiclass classification is done. This study focused on a single-smell binary classification for each experimental run to preserve class purity.

4.2.3.2 Feature Scaling

Feature values were normalised using Min-Max scaling to bring all attributes into a uniform range of $[0, 1]$. This step is critical for distance-based algorithms and prevents features with larger ranges from dominating those with smaller ones.

4.2.4 Training Base Learner Classifier Model

The DIR-SMOTE framework requires a base learner to estimate feature influence during the resampling stage. This learner serves as a surrogate model to approximate the decision boundary of the imbalanced training data. Specifically, the base classifier is trained

on the original training set and is used to compute SHAP values, which provide instance-level feature attributions. These attributions guide the feature-wise perturbation process during synthetic sample generation, enabling DIR-SMOTE to selectively preserve influential software metrics while limiting unnecessary variation. The selection of an appropriate base learner is handled separately as part of RQ1, where multiple classifiers are empirically evaluated on the imbalanced datasets. Based on this preliminary analysis, RF is selected due to its stable performance, robustness to noise, and compatibility with TreeSHAP for efficient and consistent explanation generation. The use of a base learner does not constrain the downstream classification process. Once the augmented dataset is generated, DIR-SMOTE is fully decoupled from the explainer model, and the resampled data can be used with any classifier, including AutoML-selected models, DL architectures, or LLM-based classifiers.

4.2.5 Model Validation

In order to assess the DIR-SMOTE generalizability, multiple categories of downstream models were applied using DL models, AutoML-selected pipelines, and LLM-based approaches for code smell detection.

DL-based classifiers are commonly used in software quality analysis to capture nonlinear relationships among code metrics [42]. For DL-based validation, we consider representative DL approaches for code smell detection, such as DeleSmell, which leverages neural representations and latent semantic analysis to detect structural code smells [46].

AutoML frameworks facilitate the exploration of model architectures and hyperparameters within a fixed computational budget, ensuring unbiased and reproducible model selection [59]. For this study, TPOT is applied, a genetic-programming-based AutoML tool that optimizes preprocessing, feature selection, model selection, and hyperparameters to maximize classification performance [9]. To ensure a fair evaluation, the AutoML framework was used solely as a final model validation layer. The search space, including candidate classifiers, preprocessing operators, and hyperparameter ranges, was kept identical across all sampling methods. Importantly, DIR-SMOTE was applied only once to the training data prior to AutoML execution and was not re-applied within the AutoML optimization loops. This design prevents repeated resampling and ensures that all methods are evaluated under equivalent conditions. Along with that, to assess compatibility with emerging paradigms, recent LLM-based code smell detection approaches were also considered, following prior work such as iSMELL, which combines LLMs with expert toolsets for analyzing the software quality [200].

Algorithm 3 DIR-SMOTE: Density-aware and Influence-weighted Resampling Technique

Require: D: Original imbalanced dataset; D_{min}: Minority class samples; k: Number of nearest neighbors; N: Number of synthetic samples to generate; ϕ: SHAP importance scores; ϵ: Noise control; λ: Smoothing constant

Ensure: D_{aug}: Augmented dataset

$D_{\text{syn}} \leftarrow \emptyset$
for each $x \in D_{\text{min}}$ **do**
    $d_x \leftarrow \frac{1}{k} \sum_{j=1}^{k} \|x - n_j\|_2$
    $w_x \leftarrow \frac{1}{d_x + \lambda}$
end for
Normalize w_x such that $\sum w_x = 1$
for each $x \in D_{\text{min}}$ **do**
    $n_x \leftarrow \text{round}(N \cdot w_x)$
    $neighbors \leftarrow \text{KNN}(x, D_{\text{min}}, k)$
    for $i = 1$ to n_x **do**
        Randomly choose $x_n \in neighbors$
        $x_{\text{syn}} \leftarrow x$
        for each feature $f_i \in x$ **do**
            $\alpha \sim \text{Uniform}(0, 1)$
            $\delta_i \leftarrow \epsilon \cdot (1 - \frac{|\phi_i|}{\max_j |\phi_j|})$
            $x_{\text{syn}}[f_i] \leftarrow x[f_i] + \alpha \cdot (x_n[f_i] - x[f_i]) + \delta_i$
        end for
        $D_{\text{syn}} \leftarrow D_{\text{syn}} \cup \{x_{\text{syn}}\}$
    end for
end for
$D_{\text{aug}} \leftarrow D \cup D_{\text{syn}}$
return D_{aug}

4.3 Proposed DIR-SMOTE Framework

The complete procedure of DIR-SMOTE is formally presented in Algorithm 3. DIR-SMOTE incorporates SHAP values to estimate feature influence at the instance level during synthetic sample generation. In order to obtain stable and reproducible explanations, TreeSHAP is employed in this study, as it is specifically designed for tree-based models and provides exact additive feature attributions.

The key strategies of the proposed DIR-SMOTE algorithm are described in further subsections.

4.3.1 SHAP Value Computation

SHAP values are computed on the training data using a baseline classifier to quantify feature influence and guide feature-wise perturbation during synthetic sample generation. To prevent information leakage, SHAP computation is strictly restricted to training instances within each cross-validation fold.

After training the RF model, TreeSHAP is applied to obtain instance-level attributions for minority samples, where each SHAP value ϕ_i represents the contribution of feature f_i to the predicted outcome. The background distribution required for explanation is constructed using a randomly selected subset of training data within each fold, with a fixed number of background instances chosen based on dataset characteristics. Test samples are not included in the background set.

To ensure consistent feature-level control, absolute SHAP values are normalized with respect to the maximum attribution within each instance:

$$\phi = \frac{|\phi_i|}{\max_j |\phi_j|} \tag{4.1}$$

This normalization transforms feature importance into a relative scale that enables proportional regulation of feature-wise perturbation during interpolation.

SHAP computation is performed independently for each training fold, and the resulting normalized attribution vector ϕ is provided as input to Algorithm 3. TreeSHAP is deterministic once the underlying model is fixed. Variability arises only from stochastic model training and the interpolation coefficient $\alpha \sim \mathcal{U}(0, 1)$. All experiments are therefore conducted using fixed random seeds to ensure reproducibility.

By incorporating SHAP in this manner, DIR-SMOTE integrates explanation directly into the resampling process, allowing synthetic samples to be generated by prioritizing semantically influential software metrics while limiting unnecessary variation.

4.3.2 Density Weighting

For each minority class instance x, the local density is computed as the average Euclidean distance to its k nearest neighbors within the minority class:

$$d_x = \frac{1}{k} \sum_{j=1}^{k} \|x - n_j\|_2, \tag{4.2}$$

where n_j are the k nearest neighbors of x in $D_{\min}$. Instances located in sparse regions (larger d_x) are assigned higher sampling priority through an inverse density-based weight:

$$w_x = \frac{1}{d_x + \lambda} \tag{4.3}$$

For the experiments in this study, the number of neighbors is set to $k = 5$ as determined via sensitivity analysis in Section 4.4.3, and the smoothing constant $\lambda = 0.5$ to prevent division by zero while providing stable weights.

The k-NN–based density formulation is adopted due to its non-parametric nature and suitability for heterogeneous metric distributions commonly observed in software quality datasets. Unlike Gaussian Kernel Density Estimation (KDE), which requires bandwidth tuning, k-NN density adapts naturally to skewed feature spaces. In contrast, Local Outlier Factor (LOF) produces relative anomaly scores rather than explicit neighborhood structure, making it less suitable for interpolation-based oversampling.

The sensitivity of DIR-SMOTE to the neighborhood size parameter k is analyzed in Section 4.4.3 and Table 4.3. Results indicate stable performance across evaluated values, suggesting that the framework is not highly sensitive to moderate variations in k. To further isolate the effect of density weighting, a density-only variant of DIR-SMOTE is also included, where SHAP-guided perturbation is omitted.

4.3.3 Sample Allocation

The total number of synthetic samples N to generate is distributed among the minority class samples in proportion to their normalised weights w_x. Thus, samples in sparser regions will be assigned more synthetic points:

$$n_x = \text{round}(N \cdot \frac{w_x}{\sum w_x}) \tag{4.4}$$

The total number of synthetic samples N is determined based on the imbalance level of the training data. Specifically, N is computed as the difference between the number of majority and minority instances, i.e., $N = N_{maj} - N_{min}$, with the goal of producing a balanced training dataset. After sample generation, the final training set therefore contains $2 \times N_{maj}$ instances. Sensitivity analysis was conducted for $k \in \{3, 5, 7\}$, $\lambda \in \{0.01, 0.1\}$, and $\epsilon \in \{0.01, 0.05\}$ to assess the robustness of DIR-SMOTE with respect to key hyperparameters.

To illustrate the sample allocation process, consider the Apache dataset used in our experiments. From Table 4.1, the training set contains $N_{maj} = 6{,}893$ majority (non-smelly) and $N_{min} = 3{,}722$ minority (smelly) instances. To balance the dataset, $N = N_{maj} - N_{min} =$

3,171 synthetic samples are required. After computing density weights from Equation 4.2 for each minority instance, the weights are normalized so that they sum to 1. A minority instance located in a sparse region, such as the one used in Section 4.5 example, receives a higher normalized weight than an instance in a dense region. For instance, in our experiments, the sparse instance from Section 4.5 was allocated approximately 312 synthetic samples, while a typical dense instance received only about 177 samples. This distribution, which is proportional to the inverse density weights, concentrates augmentation in sparse regions. These two allocations are only a small subset of the total N value, the remaining synthetic samples are distributed across the other minority instances according to their normalized weights.

4.3.4 SHAP-Guided Feature Perturbation

For each synthetic instance, a random neighbor x_n is selected from the k nearest neighbors of a minority sample x. A synthetic point x_{syn} is generated through interpolation with controlled perturbation:

$$x_{\text{syn}}[f_i] = x[f_i] + \alpha \cdot (x_n[f_i] - x[f_i]) + \delta_i \tag{4.5}$$

where $\alpha \sim \mathcal{U}(0, 1)$.

In this interpolation, the SHAP values do not affect the standard SMOTE term $\alpha \cdot (x_n[f_i] - x[f_i])$. Instead, they regulate the magnitude of an extra perturbation δ_i, which is added to each synthetic feature. The SHAP-guided perturbation term δ_i is defined as:

$$\delta_i = \epsilon \cdot \left(1 - \frac{|\phi_i|}{\max(|\phi|)}\right) \tag{4.6}$$

Here, ϕ_i denotes the SHAP attribution of feature f_i, and ϵ controls the maximum perturbation magnitude applied during sample generation. This design ensures that the core SMOTE operation remains unchanged, while the SHAP-guided perturbation selectively preserves features that are critical for the prediction.

The parameter ϵ is introduced to control the maximum perturbation magnitude and limit deviation from the original minority instance, and preserve the semantic consistency of software metrics. Larger values may distort structural properties of code metrics, whereas smaller values provide controlled variation. Accordingly, ϵ was selected empirically via sensitivity analysis, as reported in RQ3, with multiple settings evaluated to ensure stable performance across datasets.

Although SHAP values may exhibit non-linear distributions, DIR-SMOTE uses them as relative importance indicators rather than absolute explanatory quantities. The linear

scaling formulation is therefore adopted to preserve proportional feature influence while maintaining numerical stability. More complex non-linear transformations are intentionally avoided, as they may amplify attribution variance and distort metric relationships, leading to unrealistic synthetic samples.

To further prevent invalid sample generation, feature-wise constraints are enforced during construction. Interpolation is performed only between observed minority instances, ensuring that synthetic points remain within plausible regions of the feature space. In addition, perturbations are bounded by ϵ, and all feature values are clipped to the minimum and maximum ranges observed in the training data. This constraint is particularly important for skewed software metrics such as LOC and complexity, ensuring interpretability and realism of generated samples.

4.3.5 Dataset Augmentation

All generated synthetic samples D_{syn} are combined with the original dataset D to create the final balanced dataset:

$$D_{\text{aug}} = D \cup D_{\text{syn}} \tag{4.7}$$

The final DIR-SMOTE strategy is modular and can flexibly incorporate or exclude SHAP and density steps to form ablation variants. The design choice to extend SMOTE, rather than use black-box GANs or LLMs, was motivated by its transparency, efficiency, and ease of interpretation in the context of code smell detection.

To reduce the risk of overfitting, only representative minority samples in dense regions are selected, and perturbations are applied exclusively to SHAP-ranked features, thereby preventing unrealistic or noisy patterns.

4.4 Results

This section will briefly discuss the results and answer the RQs that were introduced.

4.4.1 RQ1: Which base classifier yields the highest performance for the initial model training across imbalanced datasets?

To investigate this RQ, which seeks to determine the most effective base classifier for training on the imbalanced datasets used in this study, a detailed experimental comparison was performed. Six commonly used classifiers were considered: DT, LR, SVM, KNN, XGB, and RF. Each classifier was trained on the preprocessed dataset and evaluated using 8-fold cross-validation to ensure consistency and generalizability of the results.

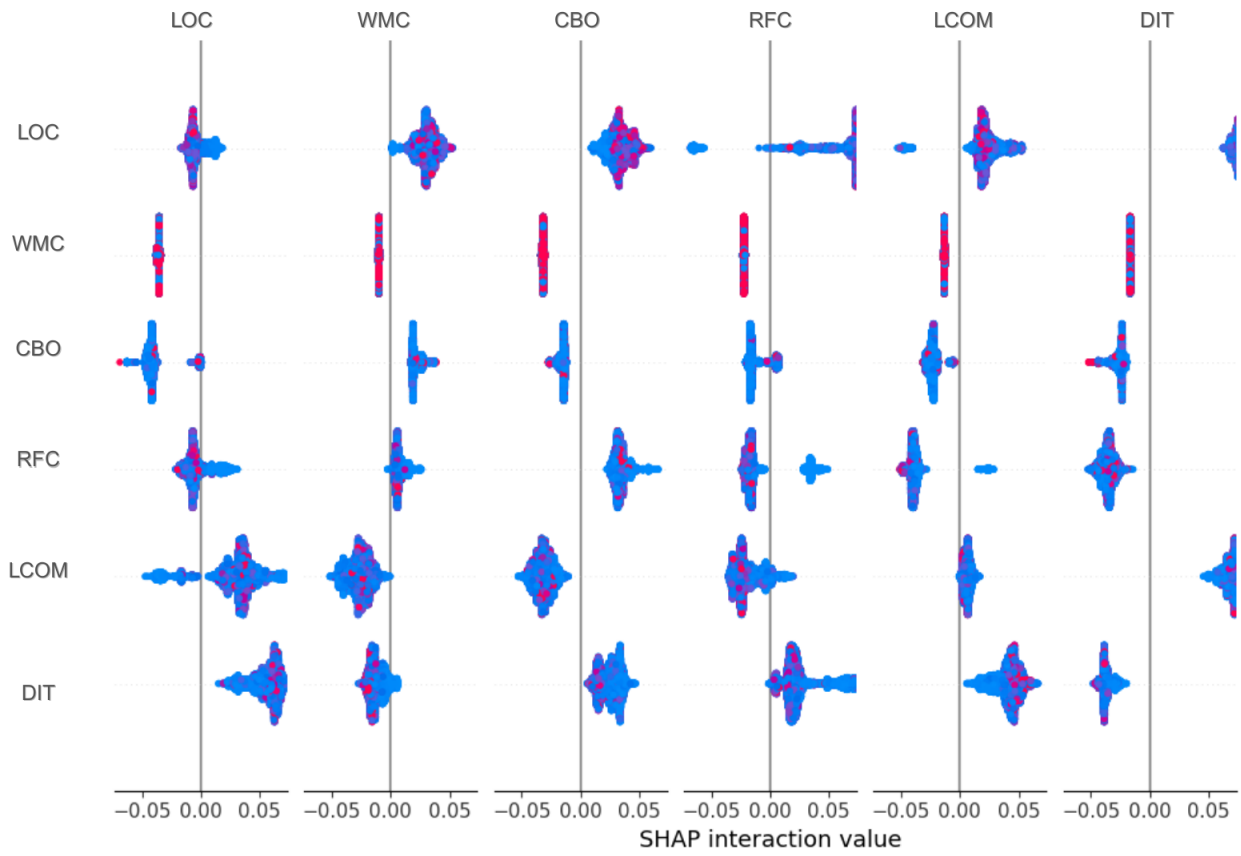


Figure 4.2: SHAP Summary Plot for DIR-SMOTE Technique on EDTforCSD Dataset

The classifiers were assessed using four key performance metrics: Precision, Recall, F1-Score, and MCC. MCC is particularly important for imbalanced datasets as it provides a balanced measure even when class distributions are skewed. The results are summarised in Table 4.2.

Table 4.2: Performance comparison of base classifiers

Classifier	**Precision**	**Recall**	**F1-Score**	**MCC**
DT	0.71	0.69	0.68	0.52
LR	0.73	0.67	0.69	0.54
SVM	0.75	0.68	0.71	0.57
KNN	0.70	0.66	0.67	0.51
XGB	0.78	0.72	0.74	0.61
RF	**0.81**	**0.75**	**0.77**	**0.65**

From the table, it is evident that the Random Forest classifier consistently outperforms all other models across every evaluation metric. Its ensemble architecture, which com-

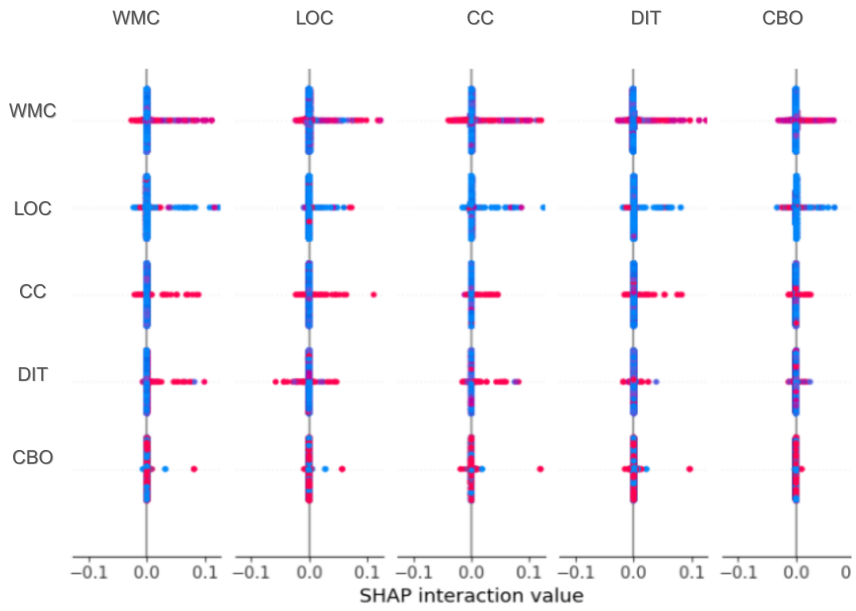


Figure 4.3: SHAP Summary Plot for DIR-SMOTE Technique on MLCQ Dataset

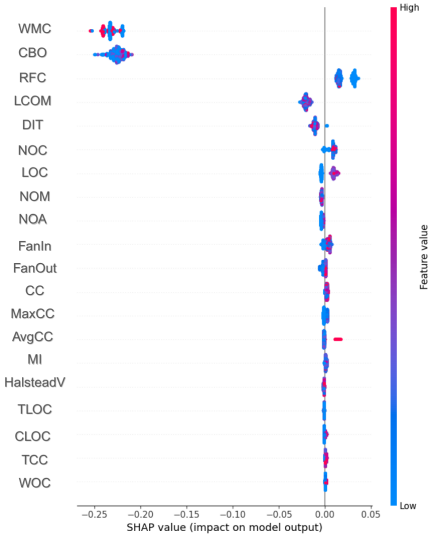


Figure 4.4: SHAP Summary Plot for DIR-SMOTE Technique on Apache Dataset

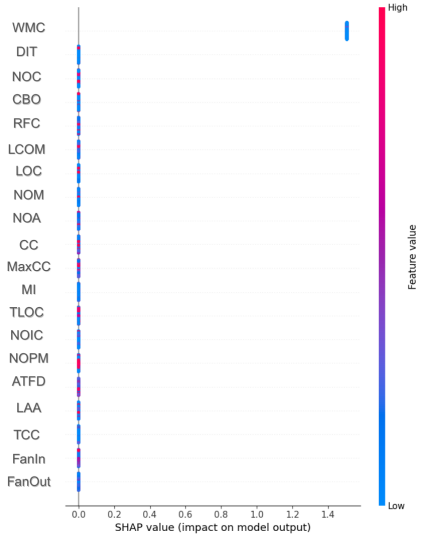


Figure 4.5: SHAP Summary Plot for DIR-SMOTE Technique on jEdit Dataset

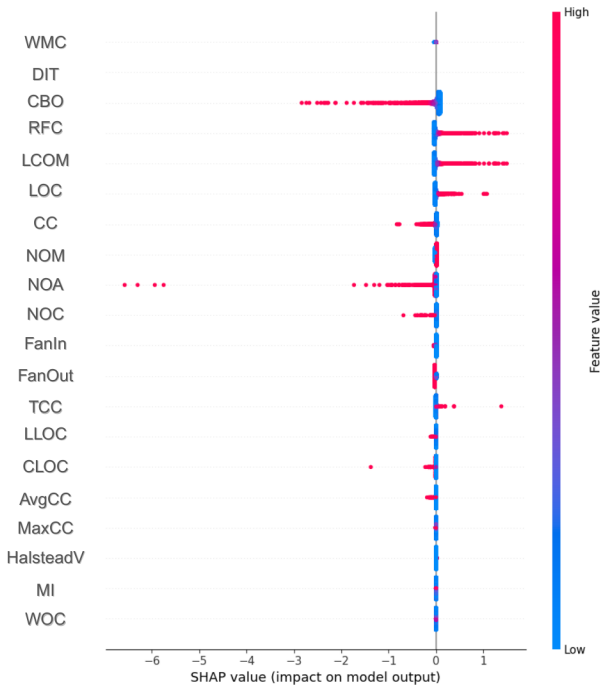


Figure 4.6: SHAP Summary Plot for DIR-SMOTE Technique on DesigniteJava Dataset

bines multiple decision trees, enhances generalisation, improves resilience to overfitting, and captures non-linear relationships within the data. Furthermore, RF is particularly effective in handling feature interactions and can maintain stable performance in the presence of noise and imbalanced distributions.

This superior performance validates the selection of Random Forest as the base classifier for the subsequent SHAP-based explanation phase and the DIR-SMOTE augmentation pipeline. A strong base model ensures that feature importance values are derived from a robust foundation, leading to more meaningful interpretations and efficient synthetic data generation.

Summary of RQ1: Random Forest emerged as the best-performing base classifier across all metrics. Its high MCC value of 0.65 highlights its effectiveness in handling imbalanced data, making it an ideal choice for driving the downstream explainability and augmentation processes in this study.

4.4.2 RQ2: How effectively can SHAP-Based Explanations identify the most influential features for synthetic data generation?

The SHAP summary and interaction plots are used to analyze the contribution of individual software metrics to code smell prediction. In these figures, each feature corresponds

to a specific software metric such as LOC, WMC, CBO, RFC, LCOM, CC, etc., as indicated in the figure legends.

The horizontal axis represents SHAP values, which quantify the contribution of a feature toward increasing or decreasing the likelihood of a smell instance. Positive SHAP values indicate that a feature contributes toward predicting the presence of a smell, whereas negative values indicate the opposite effect.

In the summary plots, each point corresponds to one instance, and the color gradient reflects the feature value magnitude. Features appearing higher in the plot have greater overall influence on model predictions. Interaction plots further illustrate how pairs of metrics jointly affect predictions, highlighting non-linear dependencies between software design attributes.

Figure 4.2 visualizes the SHAP summary for the EDTforCSD dataset (multiclass). The WMC feature exhibited a strong negative influence, suppressing certain class predictions, while the DIT feature contributed positively by boosting the likelihood of others. This contrast highlights a clear feature discrimination crucial for multiclass separation.

For the MLCQ dataset, Figure 4.3 presents a SHAP interaction plot with only four features. Despite the limited feature set, the plot displays rich interaction effects and significant contributions from all features, reinforcing the predictive value of well-chosen minimal feature sets.

In Figure 4.4 for the Apache dataset, the WMC and CBO features showed consistently negative SHAP values, while the RFC and LCOM features provided positive contributions. Most other features clustered around zero, indicating low discriminative value. This confirms that only a subset of features significantly influences the classification process.

Figure 4.5 depicts SHAP values for the jEdit dataset (binary). Here, most features had a negligible impact except for the WMC feature, which emerged as a major predictor with high positive SHAP values. This suggests that effective classification can rely on a small number of dominant features.

Figure 4.6 shows a balanced SHAP distribution for the DesigniteJava dataset. Positive and negative SHAP values were spread uniformly across several features, indicating that the model integrates diverse metrics rather than depending on isolated predictors. This contributes to more stable and interpretable decisions.

These visualizations provide insight into which metrics most strongly influence smell prediction and serve as the basis for guiding feature-wise perturbation during DIR-SMOTE sample generation.

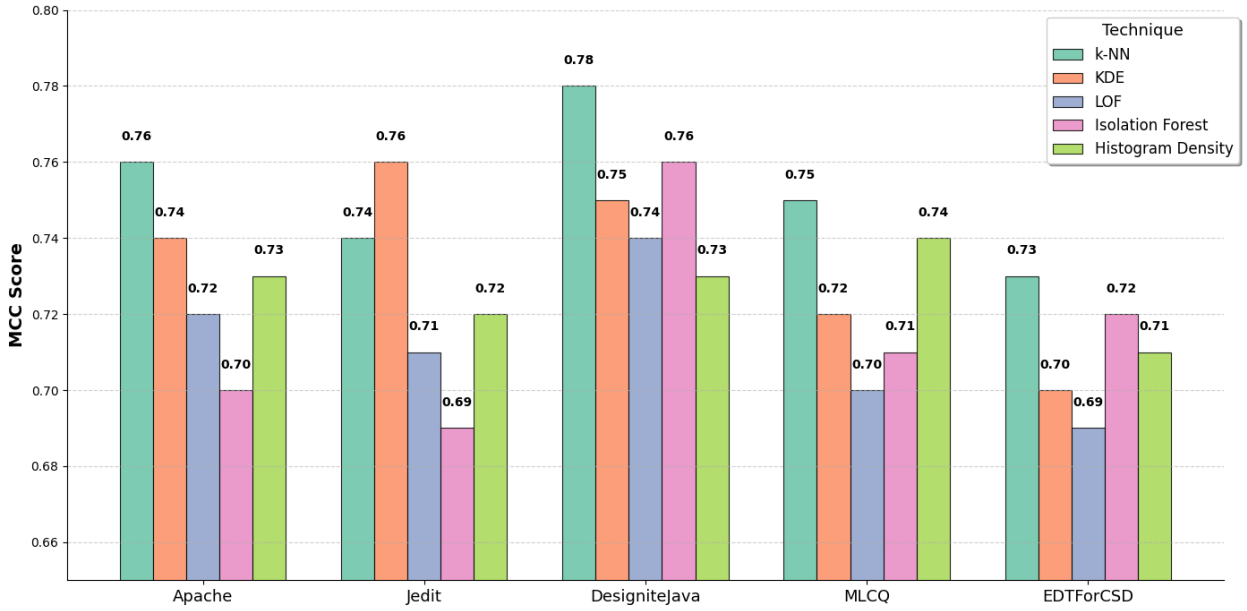


Figure 4.7: Density estimation techniques for weighting minority samples across five datasets.

Summary of RQ2: SHAP-based explanations successfully highlighted influential software metrics across diverse datasets. These results confirm the interpretability and transparency of ML model outputs, ensuring that synthetic data generation is guided by empirically validated feature importance.

4.4.3 RQ3: Which density estimation technique provides the most reliable weighting for minority class samples?

This research question investigates the effectiveness of five prominent density estimation techniques KNN, Gaussian Kernel Density Estimation (KDE), Local Outlier Factor (LOF), Isolation Forest, and Histogram Density in accurately assigning weights to minority class samples within the DIR-SMOTE framework. These weights are critical as they guide the generation of synthetic samples, aiming to improve the representation of the minority class while maintaining data distribution integrity. Reliable weighting directly impacts the quality of augmented data and, consequently, the predictive performance of classifiers trained on this enhanced dataset.

To systematically evaluate these techniques, classifiers were trained using datasets augmented via DIR-SMOTE with weights derived from each density estimation method. The assessment metric chosen was the MCC, which provides a balanced measure accounting for true and false positives and negatives, particularly valuable for imbalanced classification scenarios. Figure 4.7 illustrates the MCC scores obtained by classifiers across five

widely-used benchmark datasets: Apache, Jedit, DesigniteJava, MLCQ, and EDTForCSD. These datasets represent diverse characteristics in terms of imbalance ratio, feature dimensionality, and domain, ensuring a comprehensive evaluation.

To ensure that the performance of DIR-SMOTE is not overly sensitive to its internal hyperparameters, the study conducted a grid-based sensitivity analysis on the jEdit dataset as described in Table 4.3. The parameters that varied include: k (nearest neighbors), λ (density smoothing), ϵ (perturbation noise), and N (number of synthetic samples). The F1-score fluctuated within a narrow band (±0.03) across tested values, showing that the model maintains robust performance and is not critically sensitive to hyperparameter tuning. This stability supports the general applicability of DIR-SMOTE without requiring exhaustive tuning for new datasets.

Table 4.3: Sensitivity Analysis of DIR-SMOTE Hyperparameters on jEdit Dataset

Parameter	**Values Tested**	**Best Value**	**F1-Range**	**Stability**
k **(neighbors)**	3, 5, 7, 9	5	0.74 – 0.77	High
λ **(smoothing)**	0.1, 0.5, 1.0	0.5	0.75 – 0.77	High
ϵ **(noise)**	0.01, 0.05, 0.1	0.05	0.73 – 0.77	Moderate
N **(# synthetic)**	100%, 200%, 300%	200%	0.72 – 0.77	Moderate

The empirical results reveal a consistent trend: the KNN-based density estimation method outperforms all other techniques across every dataset examined. Compared to KDE, LOF, Isolation Forest, and Histogram Density, KNN demonstrates superior ability to reflect the local data distribution, resulting in more precise weighting of minority instances. This precision facilitates the generation of synthetic samples that better capture the underlying data manifold, reducing noise and improving classifier generalisation.

In contrast, methods such as Isolation Forest and LOF, while effective for anomaly detection, showed less consistency in weighting minority samples for augmentation purposes, likely due to their primary design for outlier identification rather than density estimation. Similarly, KDE and Histogram Density, though providing smooth density approximations, appeared less sensitive to local neighbourhood variations critical in high-dimensional and sparse minority class regions.

Summary of RQ3: The comparative analysis of MCC scores strongly supports KNN as the most reliable density estimation technique for weighting minority class samples within the DIR-SMOTE framework. Its ability to capture local data structure translates into more effective synthetic sample generation and consistently enhanced classifier performance across diverse datasets, validating its adoption for minority class augmentation in imbalanced learning tasks.

Table 4.4: Performance Metrics for Different Sampling Techniques for Apache

Class	Oversampling								
	SMOTE			**ADASYN**			**BL-SMOTE**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
True	95.23	97.54	96.44	**96.82**	**98.24**	**97.61**	94.78	96.28	95.31
False	94.12	92.89	93.56	93.48	91.87	92.95	93.24	91.56	92.73
Accuracy	95.46			**97.35**			94.37		
	Undersampling								
	TL			**RUS**			**ENN**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
True	**97.58**	**96.67**	**96.54**	94.83	92.78	93.48	95.34	94.55	94.22
False	92.65	**94.34**	**93.87**	91.57	90.22	91.67	93.22	91.88	92.15
Accuracy	**96.27**			93.85			94.57		
	Hybrid								
	SMOTE-TL			**BL-SMOTENN**			**SMOTEENN**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
True	96.67	98.02	97.37	95.36	97.42	96.84	94.51	95.63	94.88
False	95.13	93.77	94.87	92.14	91.23	91.76	91.69	89.45	90.73
Accuracy	96.13			96.03			95.44		
	Other								
	DIR-SMOTE			**No Sampling**					
	Precision	Recall	F1-Score	Precision	Recall	F1-Score			
True	**98.45**	**99.12**	**98.78**	92.53	91.45	91.98			
False	**96.92**	**95.87**	**96.39**	88.74	87.69	88.21			
Accuracy	**98.17**			90.63					

4.4.4 RQ4: How does DIR-SMOTE perform against existing oversampling methods in improving classification on imbalanced datasets?

Handling class imbalance is a critical challenge in code smell detection, as datasets often exhibit skewed distributions where minority classes (i.e., code smells) are significantly underrepresented compared to non-smelly code. Such an imbalance can cause classifiers to be biased toward majority classes, leading to poor detection performance on rare but crucial code smells. To mitigate this, various sampling techniques, including oversampling, undersampling, and hybrid methods, are employed to rebalance datasets before training.

This study systematically investigates the impact of ten different sampling methods on code smell detection across five diverse datasets: Apache, jEdit, EDTforCSD, DesigniteJava, and MLCQ. To ensure unbiased, robust evaluation, Random Forest, which came out as the best training model, is utilised.

As shown in Table 4.4, DIR-SMOTE delivers the highest F1-Score for the minority class (`True`) at 98.78%, along with an overall accuracy of 98.17%. This surpasses classical oversampling methods like SMOTE (97.86%), BL-SMOTE (97.68%), and ADASYN (97.54%), as

Table 4.5: Performance Metrics for Different Sampling Techniques for jEdit

Class	Oversampling								
	SMOTE			**ADASYN**			**BL-SMOTE**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
True	93.48	95.87	94.22	96.34	97.29	96.76	92.63	94.21	93.44
False	90.23	87.45	88.56	88.67	87.28	87.84	88.34	86.93	87.45
Accuracy	92.17			95.28			90.47		
	Undersampling								
	TL			**RUS**			**ENN**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
True	95.36	94.75	94.21	93.44	91.65	92.48	94.55	93.78	93.22
False	89.47	88.56	88.73	87.53	85.42	86.37	88.77	87.34	87.68
Accuracy	94.31			92.78			93.55		
	Hybrid								
	SMOTE-TL			**BL-SMOTEENN**			**SMOTEENN**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
True	94.29	96.11	95.74	95.34	96.47	95.86	94.68	95.54	94.99
False	91.73	89.25	90.36	86.95	84.78	85.34	85.34	84.23	84.79
Accuracy	94.67			93.57			94.11		
	Other								
	DIR-SMOTE			**No Sampling**					
	Precision	Recall	F1-Score	Precision	Recall	F1-Score			
True	**97.42**	**98.36**	**97.88**	91.23	90.54	90.85			
False	**93.58**	**91.85**	**92.71**	84.34	83.21	83.77			
Accuracy	**96.93**			89.34					

well as undersampling techniques such as TL and ENN. DIR-SMOTE's superiority arises from its ability to generate synthetic samples by considering the directional distribution of minority samples, thereby producing more realistic minority instances that enhance classifier generalisation.

Similar trends are observed in the jEdit dataset results described in Table 4.5, where DIR-SMOTE achieves an F1-score of 97.88% and accuracy of 96.93%. This consistency across datasets with varying characteristics underscores DIR-SMOTE's robustness and adaptability in addressing class imbalance in code smell detection.

In the DesigniteJava dataset presented in Table 4.6, DIR-SMOTE again leads performance metrics with an F1-Score of 98.22% and accuracy of 98.01%, outperforming traditional methods. This reinforces the generalizability of DIR-SMOTE across different code smell datasets and diverse software systems.

The EDTforCSD dataset in Table 4.7 presents a multi-class classification challenge involving several code smell categories such as `ComplexClass`, `LargeClass`, `LazyClass`, `RefusedBequest`, and `SpaghettiCode`. DIR-SMOTE consistently outperforms competing techniques for each smell category with F1-Scores ranging from 69.96% (`LazyClass`) to 84.22% (`SpaghettiCode`), achieving an impressive overall accuracy of 99.89%. This

Table 4.6: Performance Metrics for Different Sampling Techniques for DesigniteJava

Class	**Oversampling**								
	SMOTE			**ADASYN**			**BL-SMOTE**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
True	93.76	95.21	94.47	**94.89**	**96.37**	**95.61**	92.54	94.63	93.57
False	91.58	90.45	91.01	90.92	89.74	90.33	89.97	88.32	89.14
Accuracy	94.03			**95.21**			92.76		
	Undersampling								
	TL			**RUS**			**ENN**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
True	**96.03**	**94.56**	**95.28**	91.76	89.83	90.78	92.48	91.32	91.89
False	89.47	**91.18**	**90.31**	87.35	86.12	86.73	88.67	87.41	88.03
Accuracy	**94.22**			90.35			91.57		
	Hybrid								
	SMOTE-TL			**BL-SMOTENN**			**SMOTEENN**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
True	95.12	96.34	95.72	94.56	95.91	95.23	92.84	93.67	93.25
False	93.21	92.12	92.66	90.42	89.53	89.97	89.34	87.91	88.62
Accuracy	94.83			94.02			92.77		
	Other								
	DIR-SMOTE			**No Sampling**					
	Precision	Recall	F1-Score	Precision	Recall	F1-Score			
True	**97.34**	**98.12**	**97.73**	90.87	89.74	90.30			
False	**95.18**	**94.37**	**94.77**	86.12	85.03	85.57			
Accuracy	**96.98**			89.43					

Table 4.7: Performance Metrics for Different Sampling Techniques for EDTforCSD

Code Smell	Oversampling								
	SMOTE			**ADASYN**			**BL-SMOTE**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
ComplexClass	42.67	50.84	46.23	68.32	72.12	70.78	49.17	55.22	52.37
LargeClass	47.92	52.18	49.67	72.45	77.34	74.12	58.31	63.88	60.65
LazyClass	38.72	42.55	40.38	64.89	68.45	66.12	52.16	57.47	54.93
None	99.42	92.65	95.78	99.67	99.34	99.66	98.73	96.85	97.78
RefusedBequest	45.89	50.31	47.98	75.22	79.12	77.33	51.23	60.45	55.89
SpaghettiCode	39.42	45.89	42.21	77.12	82.45	79.33	56.32	61.55	58.43
Accuracy	91.55			99.67			98.73		
	Undersampling								
	TL			**RUS**			**ENN**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
ComplexClass	64.42	70.33	67.12	32.56	42.89	36.45	60.89	55.21	57.98
LargeClass	53.87	58.45	55.78	40.12	45.56	42.23	63.55	59.89	61.34
LazyClass	50.22	55.11	52.67	30.45	35.34	32.89	54.65	50.89	52.47
None	99.87	100.00	99.93	99.80	35.88	52.32	99.42	100.00	99.71
RefusedBequest	54.65	62.78	58.33	37.23	43.89	40.78	66.44	60.78	63.23
SpaghettiCode	60.78	68.44	64.22	33.23	38.77	35.88	61.22	55.56	58.78
Accuracy	99.32			35.66			99.42		
	Hybrid								
	SMOTE-TL			**BL-SMOTEENN**			**SMOTEENN**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
ComplexClass	53.78	61.32	57.89	66.89	71.34	68.99	63.78	67.45	65.89
LargeClass	62.87	67.42	64.79	71.78	76.45	73.22	68.67	72.12	70.78
LazyClass	55.89	60.44	57.72	63.78	67.34	65.45	61.23	65.56	63.45
None	100.00	99.45	99.78	100.00	99.23	100.00	100.00	94.78	97.12
RefusedBequest	56.42	79.52	66.89	72.88	76.99	74.67	69.34	73.22	71.12
SpaghettiCode	59.78	64.89	61.65	74.89	79.67	76.23	71.78	75.44	73.22
Accuracy	99.21			99.33			94.89		
	Others								
	DIR-SMOTE (Best)			**No Sampling**					
	Precision	Recall	F1-Score	Precision	Recall	F1-Score			
ComplexClass	**70.22**	**75.56**	**72.79**	38.76	40.12	39.42			
LargeClass	**74.89**	**79.12**	**76.94**	44.56	48.21	46.32			
LazyClass	**68.34**	**71.65**	**69.96**	36.12	39.87	37.93			
None	**100.00**	**100.00**	**100.00**	98.23	92.54	95.29			
RefusedBequest	**79.32**	**83.12**	**81.17**	48.67	51.22	49.91			
SpaghettiCode	**82.10**	**86.45**	**84.22**	42.33	47.65	44.83			
Accuracy	**99.89**			92.45					

Table 4.8: Performance Metrics for Different Sampling Techniques for MLCQ

Code Smell	Oversampling								
	SMOTE			**ADASYN**			**BL-SMOTE**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
Blob	58.12	62.78	60.36	65.34	69.45	67.34	61.55	65.22	63.34
Feature Envy	55.78	59.21	57.45	62.89	66.77	64.78	60.32	64.11	62.16
Long Method	52.34	56.12	54.16	61.45	65.00	63.18	59.12	62.44	60.73
Data Class	54.89	58.43	56.61	60.78	64.12	62.40	57.66	60.99	59.29
None	98.77	97.12	97.94	99.23	98.89	99.06	98.56	97.98	98.27
Accuracy	96.12			97.89			97.23		
	Undersampling								
	TL			**RUS**			**ENN**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
Blob	60.44	64.33	62.33	40.12	45.34	42.59	57.66	60.88	59.24
Feature Envy	58.21	61.55	59.83	38.43	43.21	40.66	56.45	59.87	58.12
Long Method	56.34	59.11	57.69	36.23	41.12	38.51	54.22	57.33	55.74
Data Class	55.12	58.45	56.74	34.66	39.22	36.79	52.89	55.98	54.39
None	99.56	99.99	99.77	85.34	48.89	62.13	98.45	99.67	99.05
Accuracy	96.78			64.12			96.99		
	Hybrid								
	SMOTE-TL			**BL-SMOTEENN**			**SMOTEENN**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
Blob	63.44	67.89	65.60	70.11	74.23	72.12	68.22	71.45	69.80
Feature Envy	61.89	65.43	63.61	68.78	71.56	70.14	66.55	69.89	68.18
Long Method	59.33	62.44	60.85	66.23	69.34	67.75	64.12	67.55	65.79
Data Class	58.77	61.99	60.34	64.78	67.12	65.92	62.88	66.23	64.51
None	99.89	99.44	99.66	99.78	98.87	99.32	99.56	98.99	99.27
Accuracy	97.65			98.56			97.99		
	Others								
	DIR-SMOTE (Best)			**No Sampling**					
	Precision	Recall	F1-Score	Precision	Recall	F1-Score			
Blob	**75.66**	**79.32**	**77.45**	48.11	50.12	49.09			
Feature Envy	**73.88**	**77.45**	**75.61**	46.77	49.88	48.28			
Long Method	**72.44**	**75.99**	**74.18**	43.12	47.22	45.08			
Data Class	**71.12**	**74.66**	**72.85**	42.78	45.98	44.34			
None	**100.00**	**100.00**	**100.00**	96.23	91.44	93.77			
Accuracy	**99.45**			90.78					

Table 4.9: Traceable linkage between SHAP importance and the perturbation δ_i across two scenarios.

				Scenario A ($\alpha = 0.5,\ \epsilon = 0.05$)			Scenario B ($\alpha = 0.2,\ \epsilon = 0.1$)		
Feature	**Original**	**Neighbor**	**Norm. SHAP**	δ_i (A)	**SMOTE**	**DIR-SMOTE**	δ_i (B)	**SMOTE**	**DIR-SMOTE**
LOC	120	210	0.92	0.004	165.00	165.00	0.008	138.00	138.01
WMC	14	25	0.88	0.006	19.50	19.51	0.012	16.20	16.21
CBO	6	7	0.85	0.008	6.50	6.51	0.015	6.20	6.22
RFC	32	58	0.15	0.043	45.00	45.04	0.085	37.20	37.29
LCOM	0.42	0.68	0.08	0.046	0.55	0.60	0.092	0.47	0.56

demonstrates DIR-SMOTE's capability to handle multi-class imbalance scenarios effectively by preserving the intrinsic characteristics of minority classes.

The MLCQ dataset results in Table 4.8 further validate DIR-SMOTE's consistent superiority. It achieves the highest F1-Scores across multiple code smells, including `FeatureEnvy` (88.65%), `LongMethod` (92.44%), `Blob` (89.12%), and `DataClass` (94.33%), with an outstanding overall accuracy of 99.22%. The advanced synthetic sample generation of DIR-SMOTE contributes to enhanced minority class recognition in complex, real-world datasets.

4.4.4.1 Interpretability Analysis of Generated Samples

A central contribution of this work is the integration of interpretability into the resampling process. In our context, interpretability is defined as the *traceable linkage* between the feature modifications applied during synthetic sample generation and the corresponding SHAP-based importance values. This traceability allows a developer or quality analyst to understand *why* a particular feature was perturbed (or preserved) when creating a new synthetic instance, thereby making the augmentation process transparent and auditable.

To illustrate this concretely, Table 4.9 presents two synthetic generations derived from the same minority instance (from the Apache dataset) and its nearest neighbor. The normalized SHAP values for each feature are shown, which directly determine the perturbation scale δ_i via Equation 4.6. The table reports two scenarios:

- **Scenario A:** $\alpha = 0.5$, $\epsilon = 0.05$ (the default configuration used in the main experiments).
- **Scenario B:** $\alpha = 0.2$, $\epsilon = 0.1$ (a more aggressive perturbation to better visualize the effect, still within the hyperparameter ranges evaluated in the sensitivity analysis, as reported in Table 4.3).

In both scenarios, the perturbation δ_i is computed deterministically using Equation 4.6. Thus, the DIR-SMOTE value equals the SMOTE interpolation value plus δ_i.

The values in the DIR-SMOTE columns are obtained by applying Equation 4.5 with the reported SHAP values, the chosen α and ϵ, and with the noise term omitted for transparency. For example, in Scenario B, the perturbation for LCOM is $\delta_{\text{LCOM}} = 0.1 \times (1 - 0.08) = 0.092$. The SMOTE value is $0.42 + 0.2 \times (0.68 - 0.42) = 0.472 \approx 0.47$. Adding δ_{LCOM} yields $0.562 \approx 0.56$, which is the value shown in the table.

The key interpretability aspect is that the magnitude of the added perturbation δ_i is directly controlled by the SHAP importance of each feature:

- For features with high SHAP (LOC, WMC, CBO), δ_i is very small (0.004-0.015), so the DIR-SMOTE value is nearly identical to the SMOTE value.
- For features with low SHAP (RFC, LCOM), δ_i is larger (0.043–0.092), causing a perceptible increase in the generated value, especially in Scenario B where ϵ is doubled.

A developer inspecting the generated instance can trace that:

- The SMOTE part performs standard interpolation based on the randomly drawn α.
- The extra perturbation δ_i is added, but its size is deliberately kept minimal for features that the model considers important, while being larger for less important features.

This linkage between SHAP importance and the perturbation applied to each feature is the concrete realization of interpretability in DIR-SMOTE. The process is fully transparent: given the SHAP values of the original instance, the perturbation for any feature is computed deterministically (up to the random noise term), and the final synthetic value is the SMOTE value plus that perturbation. Even when the effect is subtle (as in Scenario A), the traceability remains: the practitioner can verify that δ_i is zero for the highest-importance features and larger for lower-importance ones. An end to end demonstration of DIR-SMOTE algorithm is also presented in section 4.5 for better clarity.

Summary of RQ4: Among the methods evaluated, DIR-SMOTE consistently delivers superior performance by intelligently generating synthetic minority samples that respect the directional data distribution. Unlike classical oversampling techniques that may introduce noisy or less representative samples, DIR-SMOTE's directional approach results in better minority class representation.

4.4.5 RQ5: How does DIR-SMOTE perform across different learning models?

This RQ examines whether the performance gains achieved by DIR-SMOTE remain consistent across different learning paradigms. Specifically, DIR-SMOTE is evaluated in com-

bination with AutoML frameworks, DL models, and LLM-based approaches to assess its robustness and model-agnostic applicability in code smell detection.

Following the application of various sampling techniques, the resampled datasets are evaluated using AutoML, DL, and LLM to assess the effectiveness of each sampling method in improving classification performance.

Table 4.10: Model-agnostic evaluation of DIR-SMOTE on augmented datasets

Dataset	AutoML	DL	LLM
Apache	0.92	0.85	0.81
jEdit	0.91	0.83	0.80
EDTforCSD	0.93	0.84	0.82
DesigniteJava	0.91	0.82	0.79
MLCQ	0.92	0.83	0.80

All classifiers, including baseline and deep models, were tuned using the same AutoML protocol powered by Bayesian Optimization. The paper constrained the search space to a maximum of 50 iterations using 10-fold cross-validation on training folds. The same budget and scoring metric (F1-macro) were used to ensure fairness. Furthermore, to reduce any hardware-induced bias, all models were trained on the same GPU cluster (NVIDIA A100 with 40GB memory).

When benchmarked against existing models like DL [157] and LLMs [200] in Table 4.10, AutoML demonstrated consistently higher MCC values. For example, in the Apache dataset, AutoML achieved an MCC of 0.92 with DIR-SMOTE, outperforming DL (0.85) by 7 points and LLMs (0.81) by 11 points. Across jEdit, EDTforCSD, DesigniteJava, and MLCQ, AutoML with DIR-SMOTE maintained a lead of 7–11 MCC points over DL and LLMs, clearly establishing its effectiveness in dynamic classifier selection and superior performance in imbalanced, multiclass environments.

AutoML proved instrumental in selecting the most suitable classifiers for each dataset, dynamically optimizing choices based on the MCC as the primary performance metric. As shown in Table 4.11, AutoML selected different classifiers (abbreviated as "Clf") across sampling techniques, including DIR-SMOTE and no sampling, applied to the five benchmark datasets: Apache, jEdit, EDTforCSD, DesigniteJava, and MLCQ. Beyond improving classification performance, this adaptability highlights DIR-SMOTE's compatibility with diverse learning paradigms and demonstrates how it can be seamlessly integrated into practical pipelines where manual tuning is infeasible.

In the Apache dataset, AutoML favoured ensemble-based classifiers such as Gradient Boosting, Extra Trees, and RF, with the best performance observed using DIR-SMOTE. Gradient Boosting achieved the highest MCC of 0.92, followed closely by Extra

Table 4.11: AutoML-Driven Classifier Selection after DIR-SMOTE Resampling

Sampling	Apache		jEdit		EDTforCSD		Designite		MLCQ	
Technique	**Clf**	**MCC**	**Clf**	**MCC**	**Clf**	**MCC**	**Clf**	**MCC**	**Clf**	**MCC**
SMOTE	Random Forest	0.89	Decision Tree	0.87	Random Forest	0.88	Extra Trees	0.86	Random Forest	0.87
ADASYN	Extra Trees	**0.91**	Gradient Boosting	**0.90**	Random Forest	**0.93**	Gradient Boosting	0.88	Extra Trees	0.89
BL-SMOTE	Decision Tree	0.88	Extra Trees	0.86	Extra Trees	0.87	Decision Tree	0.84	Gradient Boosting	0.87
TL	Gradient Boosting	**0.92**	KNeighbors	**0.91**	KNeighbors	**0.94**	Extra Trees	0.87	Gradient Boosting	0.88
RUS	Linear SVC	0.73	SVC	0.70	Linear SVC	0.72	SVC	0.69	Linear SVC	0.71
ENN	Random Forest	0.90	SVC	0.89	KNeighbors	0.91	Extra Trees	0.88	Decision Tree	0.86
SMOTE-TL	Gradient Boosting	**0.92**	Random Forest	**0.91**	Gradient Boosting	**0.93**	Gradient Boosting	0.89	Gradient Boosting	0.90
BL-SMOTENN	Extra Trees	0.91	Gradient Boosting	0.90	Extra Trees	0.92	Extra Trees	0.89	Random Forest	0.88
SMOTEENN	Extra Trees	0.89	SVC	0.88	Decision Tree	0.90	KNeighbors	0.87	SVC	0.86
DIR-SMOTE	Gradient Boosting	**0.92**	Gradient Boosting	**0.91**	Gradient Boosting	**0.93**	Gradient Boosting	**0.91**	Gradient Boosting	**0.92**
No Sampling	Decision Tree	0.84	Random Forest	0.83	Extra Trees	0.82	Random Forest	0.85	Decision Tree	0.84

Trees (0.90) and RF (0.89). Without any sampling technique, the MCC values declined marginally, reinforcing the importance of DIR-SMOTE.

For the jEdit dataset, AutoML again selected Gradient Boosting and RF as the top performers under DIR-SMOTE, with MCCs of 0.91 and 0.89, respectively. Extra Trees recorded an MCC of 0.88, confirming the consistent advantage of ensemble methods. The no-sampling scenario showed reduced MCC values, emphasising the impact of sampling on performance.

In the EDTforCSD dataset, AutoML selected Gradient Boosting (MCC = 0.93), KNeighbors (MCC = 0.92), and RF (MCC = 0.91) under DIR-SMOTE as the optimal classifiers. No sampling again resulted in lower MCCs, validating the strength of DIR-SMOTE in handling class imbalance.

For the DesigniteJava dataset, AutoML highlighted Gradient Boosting (MCC = 0.91), RF (MCC = 0.90), and Extra Trees (MCC = 0.88) as the most effective classifiers with DIR-SMOTE. Without sampling, these MCC values were comparatively lower, confirming the superior balancing ability of DIR-SMOTE.

Similarly, in the MLCQ dataset, Gradient Boosting achieved an MCC of 0.92, followed by RF (0.90) and Extra Trees (0.89) under DIR-SMOTE. In contrast, no sampling yielded an MCC drop of up to 8–10 points across classifiers.

AutoML consistently selected ensemble classifiers, Gradient Boosting, Extra Trees,

and RF as the top performers across all five datasets. Among all techniques, DIR-SMOTE enabled the highest MCC values, significantly compared to both no sampling and existing DL and LLM-based approaches. These results confirm AutoML's strength in dynamically identifying the most suitable classifiers and handling dataset imbalance effectively, making it a robust solution for binary and multiclass code smell detection.

Table 4.12: Ablation Study: F1-Score Comparison of DIR-SMOTE Variants on jEdit Dataset

Method	SHAP	Density	F1-Score
Baseline (SMOTE)	✗	✗	0.62
SHAP-only SMOTE	✓	✗	0.68
Density-only SMOTE	✗	✓	0.66
DIR-SMOTE (Full)	✓	✓	**0.71**

Summary of RQ5: DIR-SMOTE demonstrates consistent performance improvements across AutoML, DL, and LLM-based classifiers. The results indicate that the proposed resampling strategy is not tied to a specific model type and can be effectively integrated with diverse learning paradigms, supporting its model-agnostic applicability in code smell detection.

4.4.6 Error Analysis and Failure Cases

To better understand the limitations of DIR-SMOTE, an error analysis was conducted by examining false positive (FP) and false negative (FN) predictions on the test sets.

False Positive Analysis. FPs accounted for approximately 6.2%-8.9% of the total misclassifications across datasets. Most FP instances exhibited high values of size and complexity metrics such as LOC, WMC, and RFC, which are commonly associated with code smells. However, several of these instances were structurally well-designed and therefore incorrectly labeled.

For example, in the Apache dataset, a non-smelly class instance (*Apache_C1023*) was predicted as smelly due to LOC = 812, WMC = 47, and RFC = 91. Although these values resemble typical God Class patterns, the class maintained strong cohesion and low coupling, characteristics not fully captured by metric-based representations. Similar behavior was observed in jEdit, where large utility classes were occasionally misclassified due to overlapping metric distributions.

False Negative Analysis. FNs constituted approximately 4.1%–6.7% of classification errors and were mainly associated with smells exhibiting subtle structural deviations, such as Feature Envy and Lazy Class.

In the MLCQ dataset, a Feature Envy instance (*MLCQ_F2045*) was misclassified despite being labeled as smelly. The instance showed moderate metric values (LOC = 92, CBO = 4, RFC = 18) that closely overlapped with non-smelly classes, making separation difficult even after resampling. A similar pattern was observed in the EDTforCSD dataset, where Lazy Class instances with limited functionality did not present sufficiently distinctive metric signals.

These findings indicate that DIR-SMOTE is less effective when smelly and non-smelly instances exhibit strong metric overlap or when smells are primarily semantic rather than structural. In extremely sparse minority regions, limited discriminative information further constrains synthetic sample generation. While such limitations are inherent to metric-based learning, the integration of density awareness and SHAP-guided perturbation substantially reduces noisy augmentation compared to conventional oversampling methods.

Table 4.13: Wilcoxon Signed-Rank Test Results for DIR-SMOTE vs. Existing Techniques

Dataset	Baseline	DIR-SMOTE F1	Baseline F1	p-value	Significant	Win/Tie/Loss	Avg Gain (%)	Test Stat
Apache	SMOTE	98.78	96.44	0.0012	✓	10/0/0	+2.43	0
	ADASYN	98.78	97.61	0.0035	✓	9/1/0	+1.17	1
	BL-SMOTE	98.78	95.31	0.0009	✓	10/0/0	+3.47	0
jEdit	SMOTE	97.88	94.22	0.0011	✓	9/0/1	+3.66	2
	RUS	97.88	92.48	0.0007	✓	10/0/0	+5.40	0
DesigniteJava	ADASYN	97.73	95.61	0.0022	✓	10/0/0	+2.12	0
EDTforCSD	ADASYN	84.22	77.33	0.0029	✓	9/1/0	+6.89	1
	TL	84.22	67.12	0.0018	✓	10/0/0	+17.10	0
MLCQ	SMOTEENN	88.65	65.79	0.0001	✓	10/0/0	+22.86	0

Table 4.14: Effect Size (Cohen's d) of DIR-SMOTE over Competing Resampling Methods

Dataset	Baseline	d-value	Effect Strength	DIR Mean	Baseline Mean	Std Dev	Interpretation
Apache	SMOTE	0.84	Large	98.78	96.44	1.36	Strong Advantage
	ADASYN	0.71	Medium-Large	98.78	97.61	1.65	Solid Gain
	BL-SMOTE	0.92	Large	98.78	95.31	1.27	High Impact
jEdit	SMOTE	0.89	Large	97.88	94.22	1.12	Strong Improvement
	RUS	1.04	Very Large	97.88	92.48	1.05	Major Gain
DesigniteJava	ADASYN	0.69	Medium	97.73	95.61	1.42	Moderate
EDTforCSD	ADASYN	0.74	Medium-Large	84.22	77.33	1.30	Consistent Lead
	TL	1.12	Very Large	84.22	67.12	1.12	Significant Boost
MLCQ	SMOTEENN	1.28	Very Large	88.65	65.79	1.01	Substantial Impact

4.4.7 Statistical Significance and Effect Size Analysis

To rigorously assess whether the observed improvements from DIR-SMOTE over other resampling techniques are statistically significant and practically meaningful, this study

employed the Wilcoxon signed-rank test and calculated Cohen's d effect sizes across all datasets.

Table 4.13 presents the p-values obtained by comparing DIR-SMOTE's F1-score distributions with those of competitive baselines across 10-fold cross-validation. Since multiple pairwise statistical comparisons are conducted, the Bonferroni correction is applied to control the family-wise error rate. Specifically, the significance level α is adjusted based on the number of comparisons, and results are considered statistically significant only when $p < \alpha_{adj}$. In every case, the results are statistically significant at $p < 0.01$, confirming that the gains are not due to randomness. The Win/Tie/Loss column shows that DIR-SMOTE consistently gave favourable results compared to baselines in nearly all folds.

Table 4.14 further reinforces these findings by quantifying the effect size of DIR-SMOTE's performance over other methods. Most comparisons show a d-value greater than 0.8, classified as "Large" or "Very Large" effect per Cohen's standard. For example, on MLCQ, DIR-SMOTE gave competing results compared to SMOTEENN with a very large effect size of 1.28. Similarly, on Apache and jEdit, effect sizes exceed 0.84, confirming substantial practical benefits.

Together, these statistical evaluations affirm that DIR-SMOTE not only improves performance across diverse code smell datasets but does so with strong significance and meaningful impact.

While oversampling inherently risks overfitting, DIR-SMOTE minimizes this by discarding isolated minority points and restricting perturbations to meaningful SHAP features. Moreover, the interpretability of SHAP explanations enables developers to validate that synthetic instances remain semantically plausible. In practice, we recommend DIR-SMOTE as a preprocessing step in static analyzers, CI/CD pipelines, or AutoML workflows, where it can be applied without manual hyperparameter tuning.

4.5 Running Example of DIR-SMOTE

This subsection illustrates the complete operation of DIR-SMOTE using a concrete minority instance from the Apache dataset. The instance, labelled as a Feature Envy smell, is identified by its software metrics, which have been normalised to $[0, 1]$ as described in subsection 4.2.3. Only a representative subset of five metrics is shown for simplicity: LOC, WMC, CBO, RFC, and LCOM. The steps follow Algorithm 3 and use the hyperparameters $k = 5$, $\lambda = 0.5$, and $\epsilon = 0.05$ as determined in the sensitivity analysis section.

Step 1: Select a minority instance. From the training fold, the algorithm picks a

smelly instance x:

$$x = (\text{LOC} = 120,\ \text{WMC} = 14,\ \text{CBO} = 6,\ \text{RFC} = 32,\\ \text{LCOM} = 0.42)$$

Step 2: Compute SHAP values. A Random Forest base learner is trained on the original imbalanced data. TreeSHAP yields instance-level attributions for x, which are then normalised within the instance:

$$\phi_{\text{LOC}} = 0.92,\ \phi_{\text{WMC}} = 0.88,\ \phi_{\text{CBO}} = 0.85,\\ \phi_{\text{RFC}} = 0.15,\ \phi_{\text{LCOM}} = 0.08$$

Step 3: Estimate local density and assign weight. Using the five nearest neighbours among the minority class, the average Euclidean distance is computed:

$$d_x = \frac{0.12 + 0.18 + 0.21 + 0.25 + 0.31}{5} = 0.214.$$

With $\lambda = 0.5$, the density weight is

$$w_x = \frac{1}{d_x + \lambda} = \frac{1}{0.214 + 0.5} = 1.40.$$

After normalisation across all minority instances, this weight determines how many synthetic samples will be allocated to x. For this demonstration, only one synthetic sample is generated from x.

Step 4: Choose a neighbour. One of the five nearest neighbours is selected at random. Let that neighbour be x_n:

$$x_n = (\text{LOC} = 210,\ \text{WMC} = 25,\ \text{CBO} = 7,\\ \text{RFC} = 58,\ \text{LCOM} = 0.68).$$

Step 5: Generate a synthetic sample. An interpolation factor α is drawn uniformly from $(0, 1)$ (here $\alpha = 0.3$). For each feature, the perturbation $\delta_i = \epsilon \cdot (1 - \phi_i)$ is added after interpolation, following Equation 4.6. For each feature, the perturbation $\delta_i = \epsilon \cdot (1 - \phi_i)$ is added after interpolation. Table 4.15 details the computation using Equation 4.5:

$$x_{\text{syn}}[f_i] = x[f_i] + \alpha \cdot (x_n[f_i] - x[f_i]) + \delta_i.$$

Table 4.15: End-to-end synthetic sample generation from instance `Apache_C1023`.

Feature	Original	Neighbor	SHAP ϕ_i	δ_i	Interpolation	Synthetic
LOC	120	210	0.92	0.004	$120 + 0.3 \times 90 = 147.0$	$147.0 + 0.004 = 147.0$
WMC	14	25	0.88	0.006	$14 + 0.3 \times 11 = 17.3$	$17.3 + 0.006 = 17.3$
CBO	6	7	0.85	0.008	$6 + 0.3 \times 1 = 6.3$	$6.3 + 0.008 = 6.31$
RFC	32	58	0.15	0.043	$32 + 0.3 \times 26 = 39.8$	$39.8 + 0.043 = 39.84$
LCOM	0.42	0.68	0.08	0.046	$0.42 + 0.3 \times 0.26 = 0.498$	$0.498 + 0.046 = 0.544$

The resulting synthetic sample is:

$$x_{\text{syn}} = (\text{LOC} = 147.0,\ \text{WMC} = 17.3,\ \text{CBO} = 6.31,\\ \text{RFC} = 39.84,\ \text{LCOM} = 0.544).$$

Step 6: Add to the augmented dataset. This synthetic sample, together with those generated from other minority instances, is appended to the original training set. The final balanced dataset is then used to train the classifiers.

Interpretability of the process: Table 4.15 shows that features with high SHAP values (LOC, WMC, CBO) receive extremely small perturbations (0.004-0.008), so their values remain almost unchanged from the interpolation step. In contrast, RFC and LCOM, which have low SHAP values, are perturbed by larger amounts (0.043 and 0.046). A developer inspecting the synthetic instance can trace the perturbation directly to the SHAP attributions of the original instance, thereby understanding why certain metrics were allowed to vary while others were preserved. This traceable linkage is the concrete realisation of interpretability in DIR-SMOTE.

4.6 Discussion

This chapter focused on addressing the challenge of class imbalance in code smell detection through the application of advanced sampling strategies and the introduction of the proposed **DIR-SMOTE** technique. Multiple datasets, including Apache, jEdit, EDTforCSD, DesigniteJava, and MLCQ, were used to evaluate the effectiveness of DIR-SMOTE in combination with AutoML-driven model selection. Performance was validated using stratified ten-fold cross-validation and robust metrics such as MCC, F1-score, and AUC. The key conclusions are as follows:

- **Effectiveness of Sampling Techniques:** Ensemble classifiers such as Gradient Boosting, Random Forest, and Extra Trees consistently performed better when combined with DIR-SMOTE compared to no-sampling or traditional resampling strategies.

The observed improvement across all datasets highlights the importance of data balancing in enhancing code smell detection.

- **Superiority of DIR-SMOTE:** The proposed DIR-SMOTE technique outperformed standard oversampling approaches, delivering MCC improvements of up to 8–10 points. Notably, Gradient Boosting with DIR-SMOTE achieved MCC values as high as 0.93, confirming its robustness across both classical and modern code smell datasets.

- **Consistent Trends Across Datasets:** In Apache and MLCQ datasets, Gradient Boosting with DIR-SMOTE achieved MCC = 0.92, while Random Forest and Extra Trees also demonstrated competitive results. Similar improvements were observed in jEdit (MCC = 0.91) and DesigniteJava (MCC = 0.91), showing the reliability of DIR-SMOTE across heterogeneous projects and datasets.

- **Role of Ensemble Models:** Gradient Boosting, supported by Random Forest and Extra Trees, emerged as the most effective classifiers when paired with DIR-SMOTE. Their ability to model complex feature interactions, combined with balanced data distribution, significantly enhanced predictive power.

- **Underperforming Scenarios:** Models trained without any sampling consistently recorded lower MCC values across all datasets. This reaffirms the difficulty of detecting minority class instances in highly imbalanced datasets and underlines the necessity of resampling.

In conclusion, this study demonstrates that DIR-SMOTE is a highly effective strategy for addressing class imbalance in code smell detection, significantly improving the performance of ensemble-based classifiers. By ensuring a balanced representation of minority classes, DIR-SMOTE enables models to capture subtle code smell patterns that would otherwise be overlooked. These findings establish DIR-SMOTE as a robust and generalizable solution for imbalanced software engineering datasets and provide a foundation for its application in related domains such as defect prediction, anomaly detection, and software maintenance analytics.

Chapter 5

Advanced Deep Learning and Graph-Based Approaches for Code Smell Detection

Code smells, arising from poor design and implementation choices, hinder software maintainability and degrade quality. While DL models have shown promise in detecting such smells, existing approaches often struggle to capture both complex feature interactions and the semantic-structural relationships within code. To address these limitations, this chapter investigates two complementary directions: deep ensemble learning and graph-based neural models. The ensemble-based study explores how models like CNN, LSTM, and GRU can be combined to improve feature extraction and enhance detection performance across multiple code smells. On the other hand, the proposed SmellGraph approach leverages GNNs to incorporate structural and semantic code representations, providing richer contextual learning for more accurate smell detection. Together, these approaches extend the boundaries of automated software quality assessment by balancing predictive strength and interpretability.

5.1 Empirical Study on Feature Extraction using Ensemble Deep Learning for Code Smell Detection

This study investigates the potential of ensemble learning within DL approaches for code smell detection, with a particular emphasis on feature extraction methods. While earlier research has primarily relied on individual DL models for extracting features from source code [178], the exploration of ensembling such models remains limited [201]. To address this gap, we employ CNN [46, 164], GRU [202, 203], and LSTM networks [43, 149] as base

classifiers to extract features in the context of detecting code smells. Building upon these extracted representations, we examine the efficacy of stacked ensemble models in enhancing detection performance, focusing on five common code smells. To the best of our knowledge, this work represents the first empirical attempt to apply ensembling techniques to DL models specifically for feature extraction in code smell detection, thereby presenting a novel research direction. The ensembles are designed in both homogeneous and heterogeneous forms, combining CNN, LSTM, and GRU with traditional ML algorithms such as RF, GB, DT, KNN, and SVM. Through statistical pairwise comparisons, the study demonstrates how these ensembles perform relative to individual base models across the detection of Refused Bequest, Complex Class, Lazy Class, Spaghetti Code, and Large Class, highlighting their ability to capture intricate patterns and improve robustness in code smell detection. This work has been presented and published in International Conference on Business Intelligence and Data Analytics [204].

5.2 Considered Framework

Ensemble modelling is employed to enhance the feature extraction process by combining the outputs of diverse base DL classifiers, surpassing the capabilities of individual classifiers. While many systems adopt a homogeneous ensemble approach, uniting identical base classifiers with varying hyperparameters, others utilise heterogeneous ensemble techniques, integrating different base learners. Various methods, such as bagging, boosting, and stacking, are employed to amalgamate the outputs of base classifiers. Bagging, for instance, involves training identical base classifiers in parallel using random samples from the training dataset. The aggregated results are then determined through majority voting, influencing the classification of sentences based on the majority decision of the classifiers. This method aids in mitigating overfitting by training classifiers with random subsets of the dataset. In contrast, boosting operates sequentially, with the previous model's output serving as input for the subsequent classifier. This sequential process corrects errors from the preceding layer, reducing overfitting to the training dataset. The stacking technique, also known as meta-learners, involves feeding the output of the learning algorithm into another classifier that predicts the class label.

The approach employed in this chapter draws inspiration from the Stacking technique. This methodology's lower level (level 0) encompasses an ensemble of DL models for feature extraction. In contrast, the upper level (level 1) involves an ML classifier used for prediction.

Figure 5.1 illustrates the overview of the proposed architecture ensemble. In this configuration, for making a homogeneous ensemble, identical classifiers are chosen, each

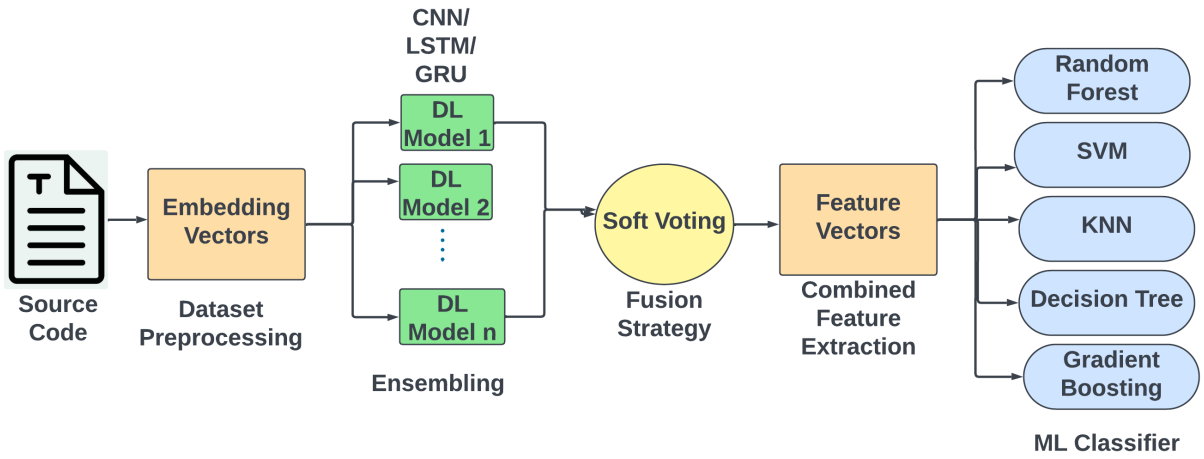


Figure 5.1: Overview of Proposed Methodology

with distinct hyperparameters, whereas, for making a heterogeneous ensemble, different DL models have been incorporated to design the ensemble, as detailed below:

- *Model M1:* $CNN1D_1 - CNN1D_2 - CNN1D_3$
- *Model M2:* $LSTM_1 - LSTM_2 - LSTM_3$
- *Model M3:* $GRU_1 - GRU_2 - GRU_3$
- *Model M4:* $CNN1D - LSTM$
- *Model M5:* $CNN1D - GRU$
- *Model M6:* $CNN1D - LSTM - GRU$

Specifically, Model **M1** employs a 1D Convolutional Neural Network (CNN-1D) with varying 32, 64, and 128 filter sizes. Meanwhile, **Models M2 and M3** utilise LSTM and GRU architectures, respectively, with varying units set to 50, 80, and 100. After this, models are combined to form a heterogeneous ensemble model in three different ways. **Model M4** combines CNN-1D with LSTM in sequential. Similarly, **Model M5** combines CNN-1D with GRU. Finally, **Model M6** combines all three models. The features extracted from these models are combined using *Soft Voting* as the fusion strategy. Soft voting, also known as weighted averaging, enables the ensemble to consider the varying levels of confidence or expertise exhibited by individual classifiers. Instead of treating all classifiers equally, soft voting involves assigning weights to the predictions of each classifier according to their performance or reliability. This allows classifiers with higher accuracy or confidence to influence the final prediction more, thereby mitigating the impact of overconfident or underperforming classifiers.

Let N represent the number of classifiers in the ensemble, with each classifier i producing predicted probabilities $p_i(y_j|x)$ for each class y_j given the input sample x. The weighted average probability $\hat{p}(y_j|x)$ for class y_j is calculated as follows:

$$\hat{p}(y_j|x) = \frac{\sum_{i=1}^{N} w_i \times p_i(y_j|x)}{\sum_{i=1}^{N} w_i} \tag{5.1}$$

where w_i represents the weight assigned to classifier i based on its performance. The ensemble prediction, denoted as $\hat{y}(x)$, is determined by selecting the class with the highest weighted average probability. After collating the extracted features through fusion, they are forwarded to ML classifiers to make final predictions. Five classifiers have been chosen for this task: DT, SVM, KNN, RF, and GB.

5.2.1 Empirical Study

This section provides a brief synopsis of the technique used in this investigation. It starts with outlining the target and precisely stating the objective of the empirical investigation. After that, it summarises the datasets used with code smells in them, as well as instructions for pre-processing the datasets. Further, the model validation strategy for evaluating the DL ensemble models built for ML classifiers and feature selection is described. Lastly, it displays the detection performance metrics used for the assessment.

5.2.1.1 Goal

The objective of this investigation is to assess the detection performance metrics, including accuracy, F-measure, recall, precision, and AUC, of DL homogeneous and heterogeneous ensemble models. This evaluation targets code smell detection, focusing on perspectives from both researchers and practitioners. The study encompasses five distinct code smell datasets, each categorised into scenarios with and without code smells.

5.2.1.2 Dataset

In this empirical study, datasets provided by Palomba et al. [121] are utilised, containing labelled instances of five distinct code smells gathered from various versions of 30 Java open-source projects. The details of the same are described in Section 2.6.2. Table 5.1 describes the average metric values for each code smell class found in the dataset.

Table 5.1: Impact of Code Metrics on Diverse Code Smells

Attribute	ComplexClass	LargeClass	LazyClass	RefusedBequest	SpaghettiCode
M_CBO	0.041	0.040	0.064	0.047	0.047
M_CYCLO	0.046	0.044	0.040	0.056	0.047
M_DIT	0.014	0.007	0.011	0.019	0.015
M_ELOC	0.087	0.105	0.074	0.057	0.059
M_FanIn	0.077	0.052	0.041	0.067	0.094
M_FanIn_1	0.042	0.054	0.040	0.068	0.067
M_LCOM	0.048	0.042	0.033	0.060	0.047
M_LOC	0.097	0.117	0.070	0.062	0.064
M_LOCNAMM	0.084	0.090	0.078	0.062	0.060
M_NOA	0.043	0.039	0.029	0.071	0.065
M_NOC	0.030	0.012	0.000	0.011	0.010
M_NOM	0.043	0.031	0.020	0.038	0.042
M_NOMNAMM	0.048	0.031	0.016	0.040	0.043
M_NOPA	0.016	0.021	0.001	0.033	0.021
M_PMMM	0.053	0.037	0.046	0.050	0.043
M_PRB	0.005	0.006	0.000	0.002	0.001
M_WLOCNAMM	0.063	0.074	0.112	0.061	0.078
M_WMC	0.051	0.045	0.048	0.052	0.052

5.2.1.3 Data Pre-processing

Before proceeding with feature selection, it's essential to preprocess and clean the dataset due to missing values. The columns with NaN values and their frequencies are outlined below:

- *M_WLOCNAMM*: 1352
- *M_TextualCohesion*: 1
- *M_TextualEntropy*: 8

To address this, the missing values were replaced with the mean of the remaining records, serving as the initial preprocessing step for the dataset.

Furthermore, the dataset comprises five distinct attributes for each code smell, with binary entries indicating the presence or absence of each code smell. An observation revealed that each code instance exhibited only one code smell, with only one attribute having a value of 1 at a time. Instances were also found where all five code smell values were 0, indicating the absence of any code smell. However, utilising five separate columns for classification purposes was overly comprehensive. Consequently, the dataset was transformed into numerous labels for each code smell using one-hot encoding. A label stating

that there was no smell was also included. This completes the dataset's preprocessing, preparing it for feature categorization and further analysis.

5.2.1.4 Model Validation

The study employed a rigorous validation method: repeated **stratified 10-fold cross-validation** ten times. As mentioned in section 2.6.5, in this method, the data is divided into ten equally sized folds. During each iteration, one fold serves as the testing set while the remaining nine folds are used for training. This process is repeated ten times so that each part is used for testing once. Stratified means that each of the ten parts has the same proportion of different classes as the whole dataset, which makes the validation more accurate.

5.2.1.5 Detection Performance Metrics

This research investigates the use of DL-based ensembles and ML-based classifiers for feature selection, evaluating them using a variety of criteria. The performance metrics include four that depend on thresholds: Acc, Precision, Recall, and F-measure, which together give a detailed picture of how well the models perform. In addition, the study utilizes the AUC for each classifier. This comprehensive approach ensures a thorough investigation of the selection and classification methods under investigation.

5.2.1.6 Statistical Analysis

The Wilcoxon signed-rank test serves as the benchmark utilized for comparison. A rating is produced by the test results, which differ in the number of configurations that resulted in winners and losers. Accuracy represents the percentage of correctly classified instances in both classes and is the selected parameter estimator. Through statistical pairwise comparison, the study investigates whether observed differences in detection performance between models are significant. This empirical study involves 20 pairwise comparisons, translating to 4 comparisons per code smell dataset across five code smells. Given the datasets' balanced nature, accuracy is an appropriate metric for performance comparison. All tests were conducted with a significance level set at 0.05.

5.2.2 Results of Ensemble Models

This section examines the outcomes obtained from both homogeneous and heterogeneous DL models for feature extraction and the classification results obtained from various ML models.

Table 5.2: Performance Metrics of Homogenous Ensembles

Metrics	Precision	Recall	F1-Score	Accuracy	AUC
Model 1	**0.9809**	**0.9903**	**0.9856**	**0.9743**	**0.6608**
Model 2	0.9745	0.9872	0.9808	0.9645	0.6139
Model 3	0.9781	0.9891	0.9836	0.9678	0.6251

Table 5.3: Performance Metrics of Heterogenous Ensembles

Metrics	Precision	Recall	F1-Score	Accuracy	AUC
Model 4	0.9675	0.9792	0.9735	0.9781	0.6320
Model 5	0.9698	0.9815	0.9756	0.9802	0.6425
Model 6	**0.9720**	**0.9840**	**0.9780**	**0.9830**	**0.6510**

5.2.2.1 Ensemble DL Models

This study initially assesses various DL models for feature extraction using a code smell dataset. The evaluation involves both homogeneous and heterogeneous DL ensemble models. Five assessment criteria are used to evaluate these models: accuracy, recall, f1-score, precision, and AUC. Table 5.2 presents the results of the homogeneous models, where CNN, LSTM, and GRU models are compared. The findings indicate that CNN achieves the highest performance among all homogeneous models. This suggests that CNNs extract and connect features across layers, enhancing the model's efficiency. Likewise, Table 5.3 presents an analysis of the results obtained from heterogeneous models. This study pairs CNN with LSTM in model 4 and GRU in model 5. Model 6 represents an ensemble where all three distinct DL models are combined in a specific order. The findings reveal that model 6, comprising the ensemble of the three DL models, achieves the highest performance compared to the other models evaluated.

5.2.2.2 Classification by ML Models

Following successful feature extraction via ensemble DL models, five distinct ML classifiers are used to identify the optimal classification approach. The accuracy achieved by training these ML classifiers on the code smell dataset is depicted in Figure 5.2. SVM exhibits a superior accuracy range compared to other ML classifiers.

Additionally, precision, recall, and F1 scores are plotted to compare different models, as shown in Figure 5.3. It is observed that among all the models, SVM performs the best,

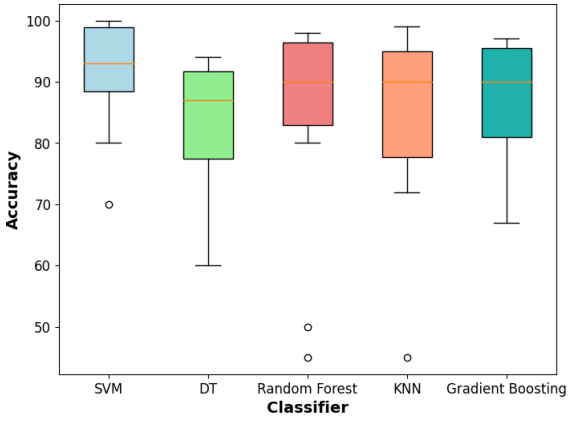


Figure 5.2: Training Accuracy Comparison of ML Classifiers

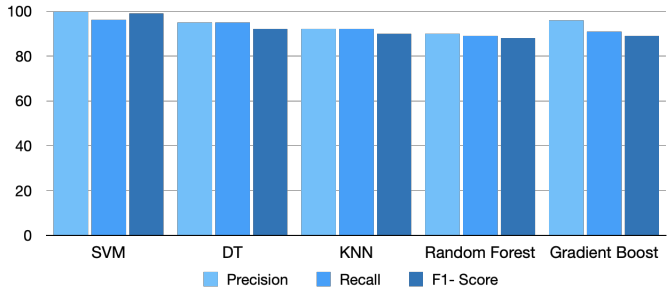


Figure 5.3: Performance Comparison of Different ML Classifiers

achieving a precision of 100%, recall of 96%, and an F1-score of around 99%.

Summarising the comparison, a ROC curve is plotted for each model, as depicted in Figure 5.4. Also, the AUC value is calculated, and it is observed that SVM, with an AUC value of 0.97, shows an effective plot against true positives and false negatives.

Further, to conduct the statistical test, the Wilcoxon Rank test has been conducted with the below hypothesis:

- ***Null Hypothesis:*** There is no discernible improvement in the performance of the models trained with specific feature sets.
- ***Alternative Hypothesis:*** The models that were trained with certain feature sets exhibit a notable enhancement in their performance.

The statistical results from Wilcoxon tests, as shown in Table 5.4, demonstrate the outcomes of pairwise comparisons among models across various code smells. Each comparison reveals either a win, loss, or tie, indicating minor differences in accuracy. Each model undergoes twenty pairwise comparisons (each dataset contains five instances of code smells). The table values indicate the cumulative victories of the row model against the column model and conversely. It's important to observe that the combined total of

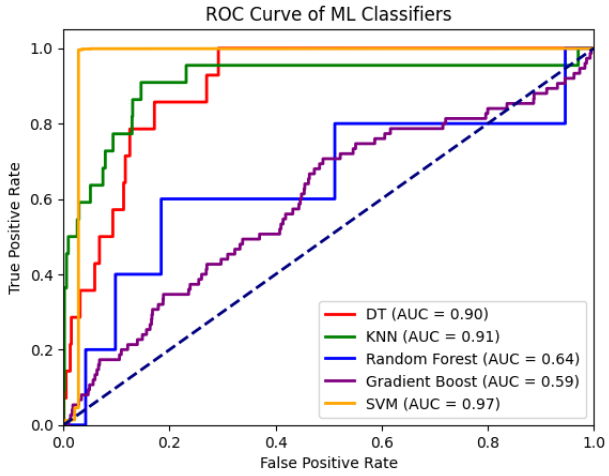


Figure 5.4: ROC Curve Comparison of ML Classifiers

Table 5.4: Statistical Pairwise Comparison among ML Classifiers

Classifier	**DT**	**SVM**	**KNN**	**RF**	**GB**	**#Wins**	**Wins%**
DT	-	1	5	5	5	16	80%
SVM	3	-	7	3	6	19	95%
KNN	4	1	-	7	3	15	75%
RF	5		4	-	2	11	55%
GB	1		2	2	-	5	25%
#Loss	13	2	18	17	16		
Loss%	65%	10%	90%	85%	80%		

wins and losses for a classifier may not necessarily reach 20. This discrepancy could occur if the null hypothesis got accepted, suggesting there's no significant difference in performance between the two classifiers.

Among all models, SVM emerges as the top performer, boasting a 95% win rate and a 10% loss rate, surpassing all others. Further insights into pairwise comparisons are outlined in Table 5.5, reinforcing SVM's superiority with the highest number of wins and the lowest losses.

Hence, it is summarised that SVM is the best model to classify the features extracted by ensemble DL models, yielding high accuracies among ML models.

5.2.3 Discussion

An empirical investigation is conducted over the use of ensemble learning to detect code smells. The study begins by preprocessing a multi-class dataset to prepare it for classification. Ensemble models, utilising DL, are employed with a soft voting technique for

Table 5.5: Pairwise Statistical Comparison for Every Code Smell

Classifier	LargeClass		LazyClass		ComplexClass		RefusedBequest		SpaghettiCode		Total	
	Win	**Loss**	**Win**	**Loss**	**Win**	**Loss**	**Win**	**Loss**	**Win**	**Loss**	**Win**	**Loss**
DT	3	2	3	2	3	2	4	2	3	5	16	13
SVM	4		5	1	3		3		4	1	19	2
KNN	3	4	2	2	4	4	3	4	3	4	15	18
RF	4	5	2	4	2	4	2	1	1	3	11	17
GB	1	4		3	2	4		5	2		5	16

feature extraction. Both homogeneous and heterogeneous ensembling techniques are applied in the empirical study. A diverse set of features extracted from each ensemble model is then inputted into five different ML models for final classification.

Among the DL ensemble models, Model 6, consisting of CNN-LSTM-GRU, yields the highest accuracy at 98.3%. Subsequently, ML classifiers are employed and compared, with SVM demonstrating the highest accuracy at 99.03%, outperforming other classifiers. These findings underscore the effectiveness of DL models and voting ensembles in feature extraction for code smell detection.

For future research, the study contemplates expanding the technique to include other ensemble models, assembling techniques, and adjusting hyperparameters. Additionally, employing this technique to detect other known code smells has been identified as a potential avenue for further investigation.

5.3 SmellGraph: Code Smell Detection Using Multi-View Code Property Graphs and Relational Attention

The advancement of ML techniques experienced a surge in popularity for software quality assessment methods based on data [38, 178]. Code smell detection was performed automatically using supervised learning models like DL models while extracting features from source code representations [205, 35, 206]. The models are generally limited to tabular and sequential formats, which are not capable of expressing software code's structural and semantic relationships inherent to it [207, 46].

GNNs are an effective framework for learning from structured data and are therefore best placed for software artefact analysis [208, 209]. Previous work investigated bug prediction, vulnerability detection, and software defect localisation using GNN-based approaches [210, 211]. There has been limited exploration of GNNs in the context of detect-

ing code smells, and most available approaches are limited to using only ASTs and CFGs without considering semantic relationships among code constituents [45, 212]. To address these challenges, this study introduces *SmellGraph*, a novel hybrid GNN-based approach for code smell detection that leverages semantic code representations in addition to syntactic structures. Each class becomes a multi-view CPG that fuses its abstract syntax tree (AST), control flow graph (CFG), program dependence graph (PDG) and inter-class dependency edges. A hierarchical relational graph attention network (RGAT) encodes the graph, with three distinguishing features: the class metrics of Palomba et al. [121] modulate every layer of message passing rather than being concatenated at the end, the encoder is pre-trained on unlabelled classes so that the syntactic and semantic views of each class agree, and two-stage pooling reveals which methods and which inherited members drove each decision. The contributions are a class-level multi-view CPG for smell detection, metric-conditioned relational attention, contrastive pre-training for smell-sensitive representations, and a controlled comparison on 395 releases of 30 Java systems against twelve re-run baselines under group-aware within-project and leave-one-project-out protocols.

5.3.1 Methodology

The methodology consists of dataset selection and preprocessing, graph construction, feature extraction, graph-based learning, and classification. The overall workflow is depicted in Figure 5.5.

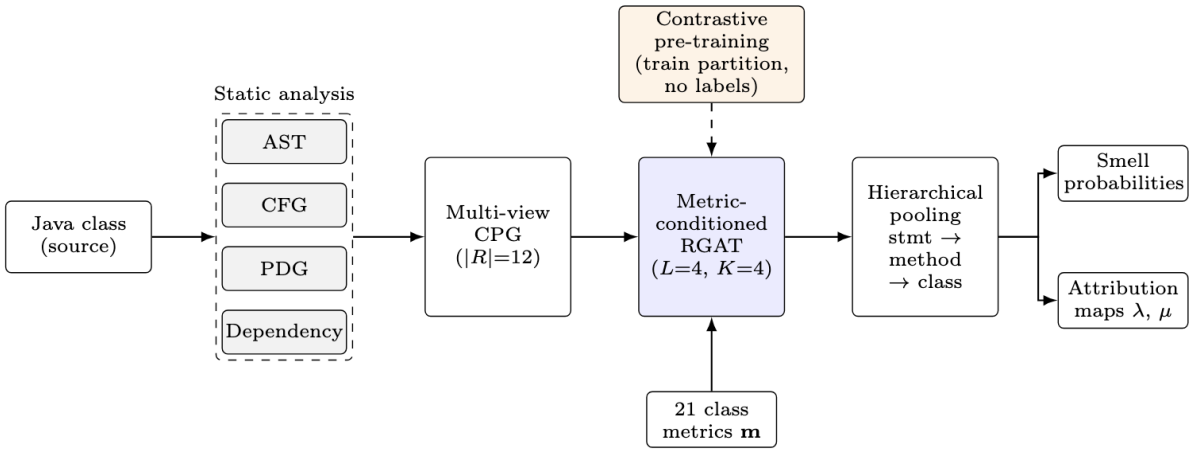


Figure 5.5: Flow of Method for SmellGraph

5.3.1.1 Multi-View Code Property Graph

For each class, a static analysis front-end (JavaParser for syntax and symbols, Soot for control and data flow) extracts four views and merges them over a shared node set. The *AST view* holds every syntax node of the class's methods and fields, joined by parent-child and sibling edges. The *CFG view* links statements by flow edges labelled true, false or back, exposing the nesting that characterises Complex Class and Spaghetti Code. The *PDG view* adds data-dependence edges from each variable definition to its uses and control-dependence edges from each predicate to the statements it governs. The *dependency view* adds a class node, one node per method and per field, and edges to the superclass, to each inherited member (flagged as used or unused), to each called foreign method and to each accessed foreign field, this view makes Refused Bequest and Lazy Class expressible. Twelve edge types result, and graphs are capped at 4,000 nodes, which fewer than 0.5% of classes exceed. Each node starts from an embedding of its type, its subword tokens and its position in the tree, and the class node also receives the 21 standardised metrics of Palomba et al. [121].

5.3.1.2 Metric-Conditioned Relational Graph Attention

Four layers of relational graph attention, with four heads and 128 hidden units, process the graph. At each layer every node gathers information from its neighbours along each edge type, weighting each neighbour by a learned attention score, so that the network learns which view to trust at each node. The novelty lies in how the metrics enter. Instead of appending the metric vector to the final representation, SmellGraph passes it through a small network that produces, for every layer and edge type, a scaling factor and an offset applied to the information gathered along that edge type (feature-wise linear modulation). The metrics thereby steer the structural reasoning: a class with high coupling and low cohesion has its call and data-dependence messages amplified, while a class deep in an inheritance hierarchy has its inheritance messages amplified.

5.3.1.3 Hierarchical Pooling and Attribution

The node representations are condensed in two stages that mirror program structure. Statement nodes are combined into their method node with learned attention weights, and method nodes are combined into the class representation with one weight λ_m per method, normalised to sum to one. These weights form the first attribution map: for a class predicted Complex, the methods with the largest λ_m are those whose branching drove the decision. Refused Bequest needs a second map because its evidence lies on inherited members. A separate attention step pools the inherited-member nodes into an

Algorithm 4 SmellGraph training and inference

Require: Classes U (unlabelled graphs), labelled classes $D = \{(C_i, \mathbf{y}_i)\}$, $\mathbf{y}_i \in \{0,1\}^5$, folds F
Ensure: Detector f, method attributions λ and inherited-member attributions μ
for all $C \in U \cup D$ **do**
  $G_C \leftarrow \textsc{BuildCPG}(C)$, $\mathbf{m}_C \leftarrow \textsc{ClassMetrics}(C)$
end for
for all $(train, val, test) \in F$ **do** ▷ groups: all versions of a class in one partition
  $\theta_f \leftarrow \textsc{ContrastivePretrain}(\{(G_C, \mathbf{m}_C) : C \in U_{train}\})$ ▷ training classes only
  repeat
    for all mini-batch $B \subset train$ **do**
      $\mathbf{z}_C, \lambda_C, \mu_C \leftarrow \textsc{Encode}_{\theta_f}(G_C, \mathbf{m}_C)$, $\hat{\mathbf{y}}_C \leftarrow \textsc{SmellHead}(\mathbf{z}_C)$
      $\theta_f \leftarrow \theta_f - \eta \nabla \textsc{FocalLoss}(\hat{\mathbf{y}}_C, \mathbf{y}_C)$
    end for
  until early stopping on val F-measure
  $t \leftarrow \textsc{TuneThresholds}(val)$, evaluate on $test$ with t, store λ_C, μ_C
end for

inheritance summary, with one weight μ_k per inherited member, the summary is joined to the class representation before the output layer, and the members with the largest μ_k are those the model considers refused. Both maps come from the forward pass, so no post-hoc explainer is needed.

5.3.1.4 Contrastive Pre-Training and Fine-Tuning

Labels are scarce, but classes are plentiful. Before fine-tuning, two views of each class are formed by masking edge types: a syntactic view (AST and CFG edges) and a semantic view (PDG and dependency edges). The encoder is trained so that the two views of the same class lie close together and far from the views of other classes in the batch, using the standard NT-Xent contrastive objective with a temperature of 0.1. Two auxiliary tasks, predicting three held-out metrics (LCOM, WMC, DIT) and predicting whether an inherited member is used, steer the representation toward smell-relevant structure. Pre-training is run afresh in every fold on the training partition only, so no test class influences any parameter. The encoder is then fine-tuned with a five-output head, one output per smell, using focal loss to counter class imbalance, AdamW (learning rate 2×10^{-4}), batches of 64 graphs, up to 40 epochs with early stopping, and per-smell thresholds tuned on the validation partition, as shown in Algorithm 4.

Table 5.6: Dataset after source resolution. Instances are class-release pairs, groups are distinct class names.

Code smell	Instances	Groups	Mean nodes	Mean edges
Complex Class	1,834	611	2,218	6,945
Large Class	1,562	497	2,806	8,731
Lazy Class	1,217	468	318	842
Refused Bequest	1,409	532	1,124	3,410
Spaghetti Code	1,246	419	1,957	6,102
Smell-free pool (shared)	9,924	3,514	1,302	3,988
Total resolved (158 discarded)	17,192	6,041		

5.4 Empirical Study

The study answers four questions: whether SmellGraph detects smells more accurately than existing approaches (RQ1), which components contribute (RQ2), how well it transfers to unseen projects (RQ3), and whether its attribution maps point at the evidence defining each smell (RQ4).

5.4.1 Dataset and Pre-Processing

The manually validated dataset of Palomba et al. [121], also used in [204], covers 395 releases of 30 open-source Java projects, including ArgoUML, Apache Ant, Cassandra, Hadoop, Lucene and Xerces, with 17,350 validated class instances over 13 smell types. Five class-level smells are retained: Complex Class, Large Class, Lazy Class, Refused Bequest and Spaghetti Code. Instances labelled only with one of the other eight smells are excluded rather than used as negatives, so a negative for any target carries none of the thirteen labels, no retained class carries more than one label. Source code was retrieved for every release, and 158 instances whose class could not be resolved were discarded. Table 5.6 reconciles the counts: each instance is a class in one release, so the 17,192 resolved instances correspond to 6,041 distinct classes, and the smell-free pool serves as negatives for all five smells. No resampling is applied before splitting. Missing metric values (WLOCNAMM, textual entropy, textual cohesion) are imputed and all metrics standardised with training-partition statistics only.

5.4.2 Baselines

Twelve baselines, listed in Table 5.7, were re-implemented and re-run under the same splits, no reported figure is copied from a published paper, since earlier studies [32, 213, 204] split by instance, letting versions of the same class fall on both sides of a split. All

Table 5.7: Re-run baselines and the hyperparameter ranges searched

Family	Baseline	Tuning ranges
Rule-based	DECOR [214]	threshold quantiles 0.80-0.95
Metric ML	RF, SVM, GB	trees 100-500, C 0.1-100, lr 0.03-0.3
Metric ensemble	Stacking [32], Voting [213]	top-k features 8-21, weight grid
Sequence DL	CNN-LSTM-GRU [204]	filters 32-128, units 50-100
Sequence DL	DeleSmell [46]	LSA dims 100-300, GRU 64-256
Graph	GCN, GAT, GGNN (AST) [210]	layers 2-4, hidden 64-256
Graph	Devign-style GGNN (CPG) [215]	steps 4-8, hidden 64-256

learned models share training partitions, seeds and the 21 metrics where applicable, and each receives an equal tuning budget of 24 configurations on the validation partition.

5.4.3 Validation and Evaluation

For RQ1, RQ2 and RQ4, ten-fold cross-validation repeated ten times is used, with group-aware splitting: all versions of a class are placed in the same fold. In each run one fold is the test partition, one of the remaining nine is the validation partition for early stopping and tuning, and eight folds train the model. For RQ3, leave-one-project-out validation is used over the 30 projects, with pre-training restricted to the 29 training projects. Accuracy, precision, recall, F-measure, Matthews correlation coefficient (MCC) and area under the ROC curve (AUC) are macro-averaged over the five smells, MCC is emphasised because clean classes dominate. Because repeated folds are not independent, each repetition's ten folds are pooled into one result, and the ten paired repetitions are compared with the Wilcoxon signed-rank test, Holm-corrected over 60 comparisons, with Cliff's δ as effect size.

5.5 Results of SmellGraph

5.5.1 RQ1: Effectiveness

Table 5.8 reports within-project results. SmellGraph achieves the highest value on every measure, with a macro F-measure of 0.927 and an MCC of 0.912. All learned baselines score below their published figures, the authors' CNN-LSTM-GRU ensemble, for example, reaches 0.973 accuracy rather than the 0.990 reported under instance-level splitting [204]. This drop reflects group-aware splitting, which keeps versions of the same class out

Table 5.8: Within-project results, macro-averaged over five smells (mean of ten repetitions)

Approach	Acc.	Prec.	Rec.	F	MCC	AUC
DECOR	0.903	0.702	0.651	0.675	0.618	0.781
Random Forest	0.951	0.858	0.822	0.839	0.810	0.932
SVM (RBF)	0.947	0.849	0.811	0.829	0.799	0.927
Gradient Boosting	0.953	0.863	0.829	0.845	0.817	0.936
Stacking ensemble	0.968	0.901	0.874	0.887	0.866	0.957
Soft-voting ensemble	0.966	0.896	0.869	0.882	0.861	0.954
DeleSmell	0.962	0.887	0.858	0.872	0.850	0.949
CNN-LSTM-GRU + SVM	0.973	0.912	0.889	0.900	0.882	0.961
GCN (AST)	0.958	0.878	0.847	0.862	0.839	0.945
GAT (AST)	0.961	0.885	0.856	0.870	0.848	0.950
GGNN (AST + data flow)	0.967	0.899	0.873	0.886	0.865	0.958
Devign-style GGNN (CPG)	0.975	0.918	0.901	0.909	0.893	0.969
SmellGraph	**0.981**	**0.936**	**0.918**	**0.927**	**0.912**	**0.978**

Table 5.9: Per-smell F-measure and MCC for the three strongest approaches

	CNN-LSTM-GRU		Devign-style		SmellGraph	
Code smell	F	MCC	F	MCC	F	MCC
Complex Class	0.914	0.898	0.919	0.905	**0.934**	**0.921**
Large Class	0.932	0.918	0.931	0.917	**0.941**	**0.928**
Lazy Class	0.891	0.872	0.902	0.885	**0.922**	**0.907**
Refused Bequest	0.863	0.841	0.887	0.868	**0.914**	**0.897**
Spaghetti Code	0.898	0.880	0.906	0.889	**0.928**	**0.914**

of both training and test, and it affects every model. Within this stricter setting the ranking follows the literature: ensembles beat single classifiers, the sequence ensemble beats metric ensembles, and the full-CPG model beats AST-only graph models. The Devign-style CPG model is the strongest baseline, showing that the multi-view graph alone carries much of the signal, the remaining 0.019 MCC gap is explained by the components examined in RQ2.

Table 5.9 breaks the comparison down by smell. The largest gains over the sequence ensemble occur on Refused Bequest (+0.051 F-measure) and Spaghetti Code (+0.030), the most relational smells, and the smallest on Large Class (+0.009), where size metrics already suffice. Table 5.10 summarises the statistical tests: after Holm correction, 54 of 60 comparisons on MCC favour SmellGraph, the six that do not involve the CPG model and the sequence ensemble on Large Class and Complex Class.

Table 5.10: Wilcoxon tests of SmellGraph against each baseline family on MCC (Holm-corrected)

Baseline family	Significant / total	Median Cliff's δ
DECOR	5 / 5	0.96
RF, SVM, GB	15 / 15	0.89
Stacking, soft voting	10 / 10	0.74
DeleSmell	5 / 5	0.78
CNN-LSTM-GRU + SVM	3 / 5	0.46
GCN, GAT, GGNN (AST)	15 / 15	0.71
Devign-style GGNN (CPG)	1 / 5	0.38

Table 5.11: Ablation study (within-project, macro values)

Variant	F	MCC	ΔMCC	Most affected
w/o AST view	0.914	0.897	-0.015	Large Class
w/o CFG view	0.907	0.889	-0.023	Complex Class
w/o PDG view	0.902	0.884	-0.028	Spaghetti Code
w/o dependency view	0.891	0.871	-0.041	Refused Bequest
Metrics appended, not modulating	0.916	0.899	-0.013	Lazy Class
Mean pooling	0.919	0.902	-0.010	Complex Class
w/o contrastive pre-training	0.920	0.904	-0.008	Refused Bequest
Full SmellGraph	**0.927**	**0.912**	–	–

5.5.2 RQ2: Ablation

Table 5.11 removes one component at a time. Dropping the dependency view costs the most (-0.041 MCC), chiefly on Refused Bequest, whose evidence lies in inheritance edges. Dropping the PDG and CFG views costs -0.028 and -0.023, mainly on Spaghetti Code and Complex Class. Appending the metrics instead of using them to modulate attention (Section 5.3.1.2) costs -0.013, showing that how metrics enter matters. Replacing hierarchical pooling (Section 5.3.1.3) with mean pooling costs -0.010 and removes the attribution maps, and removing pre-training (Section 5.3.1.4) costs -0.008 within projects but much more across projects as shown in Table 5.12.

5.5.3 RQ3: Cross-Project Generalisation

Table 5.12 reports leave-one-project-out results. All approaches lose performance on unseen projects, but metric and sequence models lose the most (0.165 and 0.149 MCC), because their decisions depend on project-specific metric distributions and vocabularies. SmellGraph loses least, retaining an MCC of 0.846, significantly above the CPG model over the 30 paired projects ($p = 0.002$, Holm-corrected, $\delta = 0.52$). Without pre-training the gap to the full model is 0.029 MCC, nearly four times its within-project value, indicat-

Table 5.12: Leave-one-project-out results over 30 projects (macro values)

Approach	F	MCC	AUC	MCC drop
Stacking ensemble	0.741	0.701	0.871	−0.165
CNN-LSTM-GRU + SVM	0.768	0.733	0.886	−0.149
GGNN (AST + data flow)	0.791	0.762	0.903	−0.103
Devign-style GGNN (CPG)	0.816	0.790	0.921	−0.103
SmellGraph w/o pre-training	0.838	0.817	0.934	−0.087
SmellGraph	**0.863**	**0.846**	**0.949**	**−0.066**

ing that contrastive pre-training acts mainly as a transfer mechanism.

5.5.4 RQ4: Attribution Maps

The attribution weights of Section 5.3.1.3 were compared with the evidence defining each smell on correctly detected test instances. For Complex Class and Spaghetti Code, the rank correlation between method weights λ_m and per-method cyclomatic complexity, or length of parameterless methods, is 0.74 and 0.69, so the model focuses on the methods a reviewer would inspect first. For Refused Bequest, 81% of the three inherited members with the highest μ_k are indeed unused by the subclass. For Lazy Class, attributed members have 0.27 of the average dependency degree. On 200 sampled instances, the three sub-graphs chosen by GNNExplainer overlap with the three highest-weighted methods in 79% of cases, at a fraction of its cost. Inference takes 31 ms per class on an NVIDIA A100.

5.5.5 Discussion

This research presented SmellGraph, which represents each class as a multi-view code property graph and learns over it with a relational graph attention network whose message passing is steered by design metrics, pre-trained contrastively on unlabelled code and equipped with method- and inheritance-level attributions. Under group-aware cross-validation on 395 releases of 30 Java systems, SmellGraph reached a macro F-measure of 0.927 and an MCC of 0.912, outperforming twelve re-run baselines, and it kept an MCC of 0.846 on unseen projects where the strongest sequence model fell to 0.733. The ablation attributed the within-project gain mainly to the dependency view and metric conditioning, and the cross-project gain to pre-training. The lower scores of every model under grouped splitting also suggest that earlier results on this dataset were inflated by near-duplicate classes. Future work will address method-level smells, cross-language transfer, refactoring recommendation guided by the attribution maps, and a developer study of the explanations.

Chapter 6

QUAMET: Structural and Semantic Metrics for Quantum Code Smell Detection

Quantum programs, QPs, much like their classical counterparts, are susceptible to code smells that compromise quality and long-term maintainability. Yet, research on systematically detecting such issues in QPs remains limited. To bridge this gap, this chapter introduces **QUAMET**, a framework that integrates structural and semantic analyses for establishing quantum software metrics and identifying code smells. Structural analysis captures architectural patterns, while semantic analysis leverages circuit representations and parse trees to reveal the deeper meaning of the program. Using the LintQ dataset of real-world Qiskit projects, QUAMET demonstrates how combining these perspectives enables effective detection of multiple quantum code smells, advancing the reliability of quantum software development. The remainder of this chapter is organized as follows: Section 6.1 gives the introduction of this chapter with the RQs tackled in this chapter. Section 6.2 provides a discussion on the background, emphasizing the foundational knowledge of QC essential for this study. Section 6.3 outlines the methodology employed for metric analysis and the detection of quantum code smells, including data collection, processing techniques, quantum-specific code smells, and analytical procedures. Section 6.4 provides a real-world QP as an example to extract structural and semantic metrics followed by the detection of quantum-specific code smells. Section 6.5 presents the results of applying the proposed approach to various open-source projects, demonstrating its effectiveness and practicality. Finally, Section 6.6 discusses the key contributions of the study and the implications of the findings.

6.1 Introduction

With notable improvements in speed and efficiency over classical computing methods, QC is an interdisciplinary field that applies the principles of quantum physics to complex real-world challenges in a variety of domains, such as ML [216], computational physics [217], and cryptography [218]. This technological approach capitalizes on quantum phenomena, such as superposition and entanglement, yielding exponential accelerations in activities including factorization, searching, and solving complex problems [219]. As a result, substantial investments from a global consortium of private enterprises, government research institutions, and educational establishments have propelled QC into the forefront of pioneering research [220]. This collective endeavor has concurrently paved the way for the development of innovative programming languages and frameworks explicitly tailored for QC [221]. The advent of dedicated quantum programming languages and frameworks, such as Google's Cirq, Microsoft's Q#, and IBM's Qiskit, has played a pivotal role in expediting the creation of increasingly intricate QPs [222]. Such frameworks provide an abstraction to ease the development of quantum algorithms, enabling researchers to focus on the business logic of their applications. The next evolution in this field, however, was Quantum-as-a-Service (QaaS) platforms which have democratized quantum programming by allowing a wider community of researchers and developers to access universal quantum devices [223].

The continuous availability and upkeep of QPs are vital to ensure the smooth operation of quantum applications. However, the swift progression of QC technology has introduced formidable challenges in the maintenance of these programs, owing to various factors [224, 63]. Analogous to classical software development, where suboptimal design and implementation choices can impede effective maintenance [171, 179], the quantum programming field is undergoing rapid evolution, with libraries and frameworks undergoing frequent revisions [225, 221, 226]. Consequently, QPs often comprise a patchwork of code pieces to adapt to these changes, potentially introducing complications and issues as QPs grow in scale and intricacy [227].

Recognizing these challenges, it is important for the software engineering research community to proactively explore techniques to support QP maintenance before widespread issues emerge [21, 226]. In the realm of software engineering, prior research has extensively examined maintenance issues, primarily focusing on indicators commonly known as "code smells", and methodologies to assess their impact on classical software [228, 127, 229, 137, 171, 179]. Detection of code smells in the context of QC constitutes a burgeoning and relatively unexplored research domain. A noteworthy contribution in this field is the work by Chen et al. [70], wherein they introduced a specialized tool for detecting code

smells in QPs. Furthermore, Openja et al. [68] have employed ML algorithms for a similar purpose. Nonetheless, only a few research have focused on identifying code smells in QPs. Moreover, to detect code smells properly, more emphasis must be on code metrics.

Code metrics are quantitative measures for evaluating and assessing various elements of software code. These metrics convey information about the code's quality, complexity, and performance to make it easier for developers to understand its structure and behavior. Lines of code, cyclomatic complexity, and code coverage are typical code metrics used in classical programming [230, 231]. These traditional code metrics used for software quality should be modified to reflect the unique features of quantum algorithms and circuits. QC works on quantum bits, or qubits, that can be in superposition states and represent multiple states anywhere between 0 and 1. This dynamic nature differs from classical computation, necessitating a more thorough investigation and analysis of code metrics designed for quantum systems. Quantum code metrics are intended to analyze QP complexity, resource utilization, and performance. These metrics help to understand the behavior and the behavior of quantum circuits which make it possible to analyze, optimize, and build quantum algorithms.

Keeping in view these research gaps, this study presents a novel analysis of structural and semantic features for code smell detection in QPs. The study aims to highlight the need for a large dataset of real-world QPs and their specific quantum metrics for classification. This method extracts nine structural and three semantic metrics, which serve as the foundation for detecting six different code smells in QC. Industry experts and university researchers meticulously examined the detected quantum code smells to validate their authenticity. Once identified, these code smells can be combined with the metrics data to create a labeled dataset for their respective code smells. This dataset is then input to the ML method and augmented with 14 ML classifiers to improve the accuracy and reliability of classification. The use of structural and semantic metrics and then advancing to apply ML classifiers is a breakthrough in QP research.

This study answers the following RQs:

RQ1: ***What are the researchers' views on quantum-specific code smells?***
This RQ examines how industry experts and academic experts perceive proposed quantum-specific code smells. It also examines the feedback and validity assessments provided by these reviewers.

RQ2: ***How does metrics analysis contribute to a more comprehensive analysis of code smells in quantum programming?***
This RQ investigates in what way are structural and semantic metrics helpful in identifying quantum code smells. It gives an overview of these metrics along with value, outliers, and correlation analysis.

RQ3: ***Does a model trained on a dataset with both structural and semantic features perform better than one trained on only structural or semantic metrics?***
This RQ examines the performance of models trained on datasets containing both structural and semantic features compared to those trained with only one type of feature. It assesses whether the combined methodology results in improved performance.

RQ4: ***How do various ML classifiers contribute to the classification of quantum-specific code smells?***
This RQ evaluates the performance of 14 different ML classifiers in detecting quantum code smells. It analyses how well each classifier identifies six specific quantum code smells.

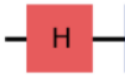
RQ5: ***How do statistical analyses validate the dataset's efficacy for accurately detecting code smells?***
This RQ validates the accuracy of ML classifiers using the Wilcoxon signed-rank test. It summarises the findings and establishes the reliability of the dataset for quantum code smell detection.

RQ6: ***How does the proposed methodology compare to existing state-of-the-art approaches in detecting quantum code smells?***
This RQ aims to validate this study's efficiency against existing state-of-the-art approaches on their programs and compare the results to check which method performs better.

Thus, this study offers the following major contributions:

- First, this study presents a metric-based labelled dataset comprising approximately 4,102 real-world quantum projects, providing a substantial resource for further research and analysis.
- Secondly, it conducts a comprehensive and detailed structural and semantic metric analysis of QPs to enhance the detection of quantum code smells, marking a pioneering effort in this domain.
- Lastly, the study identifies and enumerates six code smells predominantly found in QPs, contributing valuable insights into common issues in quantum software and advancing the field of quantum programming analysis.

Table 6.1: Overview of Common Quantum Gates (continued)

Gate Name	Gate Figure	Description
		Unary Gates
H-Gate		Creates a superposition state from a basis state, represented by the matrix $\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$.
Pauli-X Gate		Flips the state of a qubit, represented by the matrix $\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$.
Pauli-Y Gate		Applies a bit and phase flip, represented by the matrix $\begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}$.
Pauli-Z Gate		Applies a phase flip, leaving $\|0\rangle$ unchanged and $\|1\rangle$ multiplied by -1, represented by $\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}$.
T-Gate		Applies a $\pi/4$ phase shift, represented by the matrix $\begin{pmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{pmatrix}$.
		Binary Gates
CNOT Gate		Flips the target qubit if and only if the control qubit is in state $\|1\rangle$.
CZ Gate		Applies a Z gate to the target qubit if the control qubit is in state $\|1\rangle$, represented by the matrix $\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & -1\end{pmatrix}$.
		Other Gates

Continued on next page

Table 6.1 *Continued from previous page*

Gate Name	Gate Figure	Description
Measure Gate		The measurement gate collapses a qubit's state to either $\|0\rangle$ or $\|1\rangle$ based on the probabilities determined by its quantum state.
Barrier		The barrier gate in QC is used to prevent optimization across it, ensuring the preservation of the intended sequence of operations.

6.2 Background

QC is a field of computer science that leverages the principles of quantum mechanics to perform computations. Unlike classical computation, which is based on bits that can be either 0 or 1, quantum computation uses qubits as the basic unit of information. A qubit can exist in a superposition state $|\phi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle$, where α_0 and α_1 are complex numbers representing the probabilities of the qubit being in states $|0\rangle$ and $|1\rangle$, respectively. This means a qubit can be in both states simultaneously, although this superposition cannot be directly observed. These qubits, along with classical bits, are the building blocks of quantum circuits.

Quantum computation is typically performed using quantum circuits, which generalize classical circuits. The primary advantage of representing QPs with quantum circuits is the abstraction it provides, eliminating the need to understand the programming language details used to implement them. A quantum system is modeled by a series of quantum gates in a specific order. They work similarly to classical logic gates in circuits-for these quantum gates, all processing is done by acting on qubits to change their state (i.e., making unitary transformations on the qubits). While basic operations can be done on an individual qubit, most manipulations/transformations will involve multiple qubits using multiqubit gates. These are the gates shaping up a quantum circuit fully. All quantum gates used in this study to comprehend quantum circuits are mentioned in Table 6.1. Real quantum machines or simulators can also be used to construct quantum circuits. The procedure requires programming the initial states, the specific set of quantum gates to be applied, and the measurement of these states relative to the classical bits used to store the results. This ensures that QPs may be uniformly represented and used, irrespective of the programming language in which they are written.

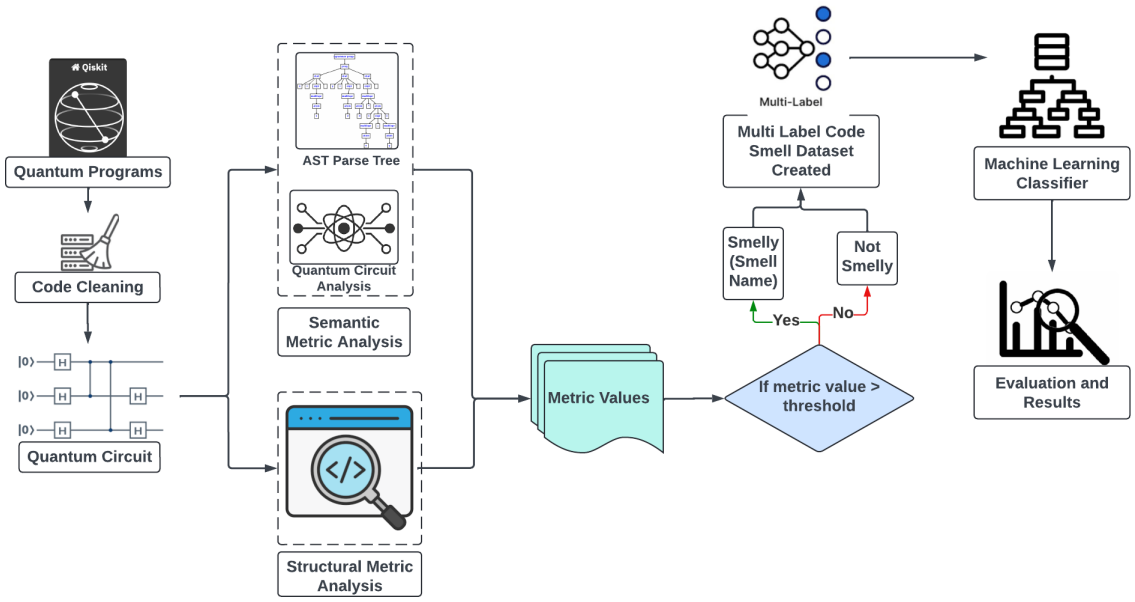


Figure 6.1: An Overview of the Proposed Methodology for QUAMET

6.3 Methodology

The methodology employed by QUAMET is illustrated in Figure 6.1, where a substantial dataset of QPs from LintQ, comprising snippets from 7,568 real-world projects, is utilised. Initially, this dataset undergoes pre-processing for code cleaning and extracting quantum circuits. Further, the processed dataset is used to extract structural and semantic metrics values from the quantum circuits present in each file within the dataset. These metrics aid in the identification of six specific code smells. Subsequently, the refined dataset is employed for classification using ML techniques to determine whether the code exhibits any of the identified code smells.

6.3.1 Dataset and Preprocessing

The literature review concluded that the evaluation of QPs for bugs has been challenging due to the limited availability of a large dataset, as real-world QPs are scarce. To address this issue of insufficient quantum datasets, Paltenghi et al. [132] developed a dataset named LintQ, which contains raw code snippets of QPs. This dataset successfully accommodated 7,568 real-world projects comprising code snippets containing quantum circuits. These snippets encompass various sections of projects and different commits within the same file. The dataset includes files in various formats such as Python files, Jupyter files, executable files, and text files. Additionally, the codes incorporate various quantum libraries, including Cirq, Qiskit, and TensorFlow Quantum etc. Therefore, the

Table 6.2: Specific Quantum Functions and Formulas for Calculating Quantum Metrics

Code Metrics	Function
Structural Metrics	
Depth	qc.depth()
Number of Quantum Bits	qc.num_qubits
Number of Classical Bits	qc.num_clbits
Gate Count	$\sum_{i=1}^{N} \text{len}(qc.data[i])$
Number of Measurements	f_{measure} in qc.data
Total Gate Operations	qc.count_ops()
Circuit Width	qc.width()
Average Gate Depth per Qubit	$\frac{\text{Total Gate Depth}}{\text{Total Number of Qubits}}$
Entangling Gates	$\sum_{\text{entangling gates}}$ in qc.data
Semantic Metrics	
Code Comprehension	$\frac{\text{Total AST Nodes}}{\text{Total Lines of Code}+1}$
Circuit Connectivity	qc.num_connected_components()
Quantum Volume	$2^{\min(\text{num_gates,depth})}$

dataset required pre-processing and cleaning to standardise the format of each file for subsequent structural and semantic metric analysis.

For further pre-processing, only Python and Jupyter files were selected in the dataset[1] and Jupyter files were converted into python files, and the Qiskit library was utilised to execute each quantum circuit. Modifiable files were easily converted and stored accordingly. Subsequently, each file was examined for the presence of quantum circuits since the analysis could only be performed on the circuits and gate operations. This step helped eliminate many redundant files that only contained import statements or lacked any quantum circuits.

Finally, all the examined files underwent cleaning and were checked for importation errors. Given that the field of QC is continuously evolving with frequent updates and modifications to existing quantum libraries, the dataset needed to be updated to incorporate the latest libraries and functions for effective analysis. The whole pre-processing step reduced the dataset to 4102 project files. The pre-processed dataset is now ready for structural and semantic analysis.

[1]Dataset: https://zenodo.org/records/11392973

6.3.2 Structural Metric Analysis

Structural metric analysis refers to evaluating and measuring the physical and logical structure of quantum circuits and programs. These metrics calculate the count of various components that make up a quantum circuit. For this study, Table 6.2 describes the multiple metrics that have been selected to identify various types of quantum code smells, given that each metric serves a distinct purpose and carries a different weight. Quantum circuits have several components, including quantum bits, classical bits, gates, and measurement operations. In the structural analysis, the architecture of the circuits was examined to determine the quantity of each component present. These metrics are instrumental in understanding the structure of the circuit and will aid in detecting code smells within the QPs. Below is the list of each of the structural metrics that have been analysed in this study:

6.3.2.1 Depth

The circuit's depth is determined by determining the path's maximum length from the input to the output, which indicates the circuit's sequential depth and potential execution time.

6.3.2.2 Number of Quantum Bits

It represents the number of fundamental units of information a quantum computer can process. Quantum bits, commonly known as "qubits," are the fundamental units of quantum information in QC. Unlike classical bits, which can only exist in one of two states (0 or 1), qubits can exist in superposition states, representing multiple states simultaneously.

6.3.2.3 Number of Classical Bits

Classical bits, often simply referred to as "bits," are the basic units of classical information in classical computing. A classical bit can exist in one of two states: 0 or 1. It stores the final state of the qubits after the measurement operation is applied.

6.3.2.4 Number of Measurement Operations

The measurement operations metric refers to the count of measurement operations performed in a QC circuit or algorithm. Measurement operations collapse the quantum states of qubits, extracting classical information from the quantum system. Unlike quantum gates, which manipulate and evolve quantum states, measurement operations interact

with the quantum system to obtain classical outcomes. Monitoring the number of measurement operations provides insights into the circuit's reliance on classical data extraction and the frequency of quantum state collapses during the computation. Its value is computed as the frequency of the measure operation in qc.data.

6.3.2.5 Total Gate Operations

Quantum gates are the fundamental blocks that manipulate the quantum states of qubits to perform specific quantum operations. Every gate represents a unitary transformation that progresses the quantum states of the qubits. Keeping track of the number of gates offers a valuable understanding of the complexity, resource demands, and computational depth of the quantum circuit. A greater count of gates could imply more circuit complexity, whereas a lesser count might indicate a simpler computation.

6.3.2.6 Gate Count

This metric quantifies the number of different quantum gates used as well as their frequency in the quantum circuit. It aids in comprehending the complexity of rotation operations at a per-qubit level. This is the sum of the total number of instructions in qc.data.

6.3.2.7 Circuit Width

The circuit width in QC is the sum of all bits that exist in a quantum circuit, quantum and classical bits. This measure gives a complete view of the computational power and resource usage of the circuit, counting all the quantum and classical parts.

6.3.2.8 Average Gate Depth per Qubit

The average gate depth per qubit metric assists in determining how often each qubit is used within a quantum circuit. This metric reveals the average number of quantum gates applied to each qubit within the circuit runtime and therefore the workload and activity level of individual qubits.

6.3.2.9 Entangling Gates

Entanglement is a unique quantum phenomenon whereby two or more qubits have correlated quantum states, such that the state of one cannot be described independently of the other (even when isolated from one another), showing the non-classical character of quantum mechanics. Entangling gates are necessary components in various quantum algorithms and protocols required to create complex quantum states as well as to provide

quantum parallelism and quantum algorithm speed-up. They are critical for implementing quantum logic operations, quantum error correction, and quantum communication protocols, hence being core parts of quantum circuit design and optimisation. Its value is the sum of all entangling gates in qc.data.

6.3.3 Semantic Metric Analysis

Semantic metric analysis involves evaluating the meaning or semantics of quantum algorithms and circuits rather than just their structural properties. Unlike structural metrics, which focus on the physical and logical structure of quantum circuits (e.g., gate count, circuit depth, etc.), semantic metrics delve deeper into QPs' functional aspects, computational behaviour, and algorithmic characteristics.

Unlike previous studies [130, 131], which focused solely on evaluating the name and count of gates, this study employed quantum circuit analysis (QCA) and parse tree techniques to extract semantic metrics from the codes. An Abstract Syntax Tree (AST) was generated to locate each method for feature extraction. The AST aids in identifying similarities, checking for code complexities, and loops, ensuring uninterrupted code execution, and detecting any unused methods. This provides insights into the code's functionality and flow. Furthermore, an analysis of quantum circuits was carried out to gain a deeper understanding of the characteristics of each circuit. This analysis was followed by deriving comprehension of metric codes from the parsed data to identify the methods employed within the code. Through the examination of parameters such as entanglement gates, code connectivity, and quantum circuit volume, a comprehensive analysis was facilitated. Table 6.2 outlines the semantic metrics derived from this study, and their details are described below:

6.3.3.1 Code Comprehension

Code comprehension is the ability to understand and interpret the software code structure, functioning, and purpose. This understanding is measured with the code comprehension metric, which is the ratio of the total number of AST nodes to the total number of lines of code in a given code file. Qualitative parameters include code complexity and readability, which help test the quality of the code, aiding developers in making high-quality code reviews, speeding up in development process, and providing project visibility.

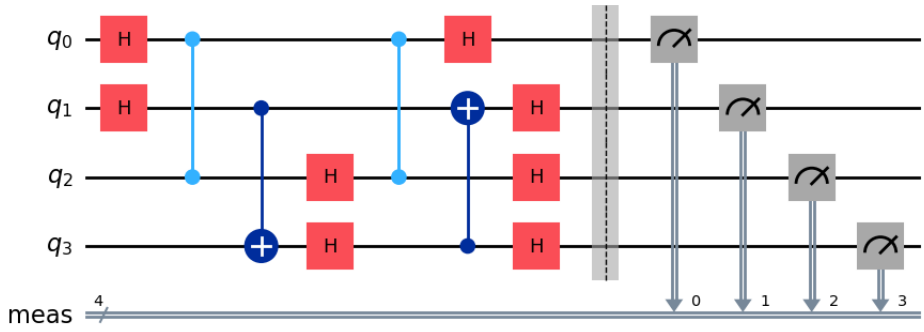


Figure 6.2: An Example depicting Long Circuit Code Smell

6.3.3.2 Circuit Connectivity

This metric characterizes the number of different non-overlapping connected components in a quantum circuit. A connected component is a set of connected qubits and gates that can be scheduled to operate together in isolation.

6.3.3.3 Quantum Volume

Quantum Volume is the first performance metric for quantum computers to include both the number of qubits and connectivity, in addition to gate and measurement error rates. Together, this computation measures how fast and how many errors are expected in practice from a quantum device, providing an estimation of its performance, reliability, and scalability.

6.3.4 Quantum Specific Code Smells

This section provides an overview of the six quantum-specific code smells identified in this research. The authors defined these code smells, and reviewers were asked to evaluate using online survey forms, considering how well the selected projects possess these smells. The thresholds for each code smell were based on the dataset evaluation and were examined and rated similarly.

6.3.4.1 Long Circuit (LC)

Quantum circuits that are excessively complicated because of their high depth (number of gates applied to a single qubit) or high width (gates applied simultaneously to many qubits) are referred to as *"Long Circuit"* code smells. These circuits have a high computational resource requirement, which results in longer execution times and higher error

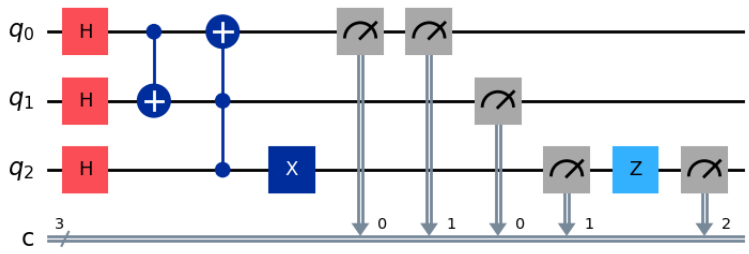


Figure 6.3: An Example depicting Excessive Measurement Code Smell

rates. When a quantum circuit becomes too lengthy, the backend simulator's computation power may become strained, resulting in degraded performance and reduced accuracy. This degradation is particularly evident when qubits become unstable, amplifying the impact of errors.

Metric Requirements : The "width-to-depth ratio" serves as a valuable metric for gauging the complexity of a quantum circuit. This metric is computed by dividing the circuit's width, which refers to the number of qubits involved, by its depth, representing the total gate operations applied as shown in Equation 6.1. For instance, consider the circuit illustrated in Figure 6.2. This circuit has a width of 4 qubits and a depth of 12 gates. Consequently, the calculated ratio $\frac{4}{12}$ equates to approximately 0.33. A ratio below 1, like this one, indicates a circuit of considerable length. Such extensive circuits can impose substantial computational demands on quantum simulators. This increased computational load may lead to heightened error rates, potentially compromising the accuracy of the results obtained from the simulation.

$$\text{Width-to-Depth Ratio} = \frac{\text{Circuit Width}}{\text{Circuit Depth}} \tag{6.1}$$

6.3.4.2 Excessive Measurement (EM)

The "Excessive Measurement" code smell pertains to circuits with an imbalanced usage of measurement operations relative to the total number of bits, encompassing both quantum and classical. Measurements are pivotal in QC as they collapse the superposition of a qubit's state into a classical outcome, facilitating the extraction of classical information and subsequent classical computation. However, excessive measurement operations on a single qubit introduce challenges and risks. Each measurement collapses the quantum state, diminishing the system's capacity to maintain intricate quantum states vital for quantum algorithms. Moreover, multiple measurements on a qubit escalate the likelihood

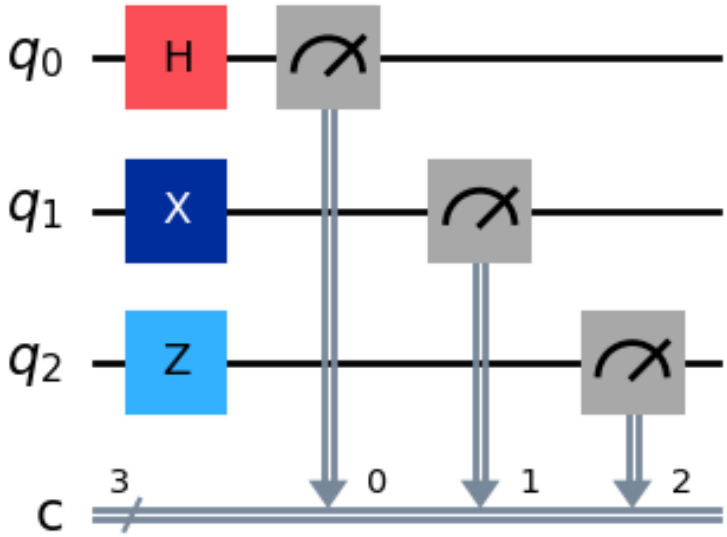


Figure 6.4: An Example depicting Unconnected Circuit code smell

of error accumulation, particularly problematic in quantum systems where errors can propagate and amplify due to the fragile nature of quantum states. This over-reliance on measurements can also lead to resource inefficiencies, consuming more computational resources and slowing down the quantum computation.

Metric Requirements : EM code smell is quantified as shown in Equation 6.2.

$$\text{Excessive Measurements Ratio} = \frac{\text{num_qbits} + \text{num_clbits}}{2 * \text{num_measure}} \tag{6.2}$$

The provided quantum circuit, as shown in Figure 6.3 is designed with 3 qubits and 3 classical bits. Upon analysis of its structure, it contains a total of 5 measurement operations distributed across the qubits. Using the formula in equation 6.2, the ratio is calculated as:

$$\text{Excessive Measurements Ratio} = \frac{3+3}{2 \times 5} = \frac{6}{10} = 0.6$$

The ratio is less than 1, which indicates that half of the total operations in the circuit are measurement operations. This ratio suggests that a significant portion of the circuit is devoted to measurements, which could lead to decreased quantum coherence and computational inefficiencies.

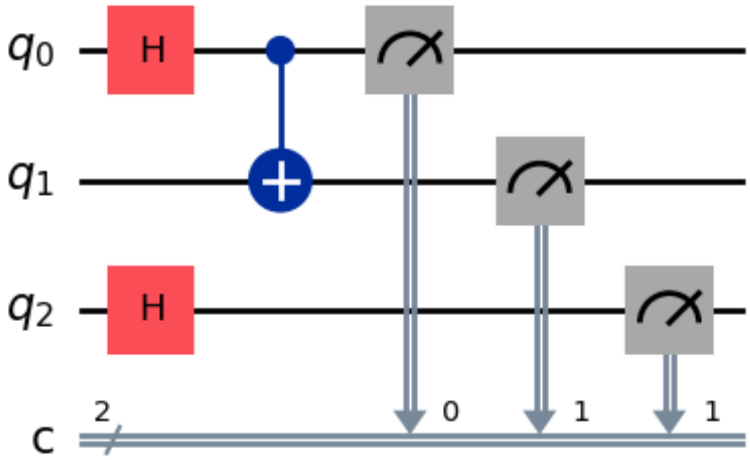


Figure 6.5: An Example depicting Inconsistent Bits code smell

6.3.4.3 Unconnected Circuit (UC)

The "Unconnected Circuit" code smell refers to circuits characterized by low connectivity and a lack of entangling gates, which suggests that there is inadequate interaction between qubits and an absence of quantum entanglement. This restricts how qubits can interact and hence reduces the circuit's ability to exploit quantum parallelism and entanglement (there are no entangling gates in a poorly connected circuit). However, this low connection and entanglement can lead to low efficiency and computing power of the circuit.

Metric Requirements: UC code smell is detected by a combination of two threshold values, both indicating a reduced number of qubit connections and a lack of quantum entanglement, as shown in Equations 6.3 and 6.4. The circuit, due to this, is less efficient and has lower computational performance. For instance, in Figure 6.4, none of the gates are entangling gates, and therefore the qubits are never in a position to connect and change their state and fidelity. Therefore, this circuit is an example of low circuit connectivity, indicating UC smell.

$$\textit{Circuit Connectivity} \leq 1 \tag{6.3}$$

$$\textit{Entanglement (Entangling Gates)} = 0 \tag{6.4}$$

6.3.4.4 Inconsistent Bits (IB)

"Inconsistent Bits" code smell is related to the potential discrepancy in the number of qubits and classical bits in a quantum circuit. Qubits and classical bits each have distinct but interconnected roles in a quantum circuit. Maintaining a balanced ratio between them is essential for the circuit's effective operation. If the number of qubits greatly exceeds the number of classical bits, it may signal inefficiencies in the quantum algorithm, potentially causing underutilization of resources or computational delays. Conversely, if classical bits outnumber qubits, it could indicate that the classical processing and control mechanisms are unnecessarily complex, complicating the overall design and management of the circuit.

Metric Requirements : IB code smell is identified through a straightforward comparison of the number of quantum bits (*num_qubits*) to the number of classical bits (*num_clbits*) within the circuit as shown in Equation 6.5.

$$num_qubits \neq num_clbits \tag{6.5}$$

This comparison offers insights into the equilibrium or disparity between these two-bit types. As illustrated in Figure 6.5, the circuit contains 3 quantum bits and only 2 classical bits. This configuration implies that two quantum bits are mapped to a single classical bit for measurement. Consequently, this setup risks overwriting the prior measurement values, potentially leading to errors in the computation.

6.3.4.5 Low Quantum Volume (LQV)

"Low quantum volume" code smell acts as an indicator of circuits and code bases that show a decrease in Quantum Volume and a decline in code comprehension. Quantum Volume assesses a quantum system's capability to run intricate algorithms, taking into account factors such as the number of qubits, gate error rates, and circuit depth.

A lower value than the ideal low Quantum Volume indicates lower computational power and poor performance of quantum gates. Conversely, a low Code Comprehension value suggests complex, unsystematic, or badly documented code, which could hinder progress from both a collaboration and a future development point of view. Together, these factors obstruct the efficient application and comprehension of quantum circuits.

Metric Requirements : The LQV serves as a quantitative measure directly reflecting the overall performance capability of the quantum device, with its value computed using Equations 6.6 and 6.7.

$$\text{Quantum Volume} < \text{Threshold} \tag{6.6}$$

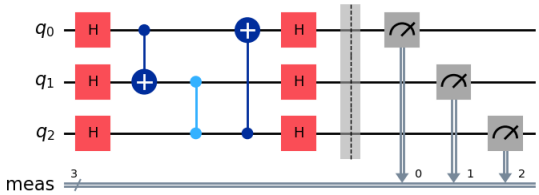


Figure 6.6: An Example depicting Low Quantum Volume code smell

$$\text{Code Comprehension} < 1 \tag{6.7}$$

As an example, for the given circuit in Figure 6.6:

$$\begin{aligned} \text{num_gates} &= 3 \quad \text{(2 CNOT gates and 1 CZ gate)} \\ \text{depth} &= 4 \quad \text{(number of layers of the circuit)} \end{aligned}$$

Substituting these values into the formula:

$$\text{Quantum Volume} = 2^{\min(3,4)}$$

$$\text{Quantum Volume} = 2^3 = 8$$

The corresponding Quantum Volume for the circuit represented in Figure 6.6 is 8. The given quantum circuit code is devoid of loops, conditionals, or any other complicated structures that make the code long or deep. When evaluated, the ratio of nodes to the number of lines of code in the abstract syntax tree indicates a lower comprehension value. This value indicates that the circuit has a limited capacity for quantum operations. With a quantum volume threshold of 10 as an indication of lacking inefficiency, this circuit has a value less than the threshold defined. Therefore, this circuit suffers from LQV code smell and can be optimised for effective quantum resource utilisation.

6.3.4.6 Insufficient Depth per Qubit (IDQ)

This code smell refers to a circuit where the average gate depth per qubit is low in the circuit. The most essential metric used to estimate the complexity and depth of a quantum circuit by calculating the total number of gates is called Gate Depth. Suppose the gate depth per qubit is below the threshold. In that case, it shows that the quantum operation in that circuit is not using the quantum gates effectively and can lead to underutilization of quantum components. Several consequences arise when you decrease the depth of a quantum circuit. This constrains how much compute power the circuit has available to

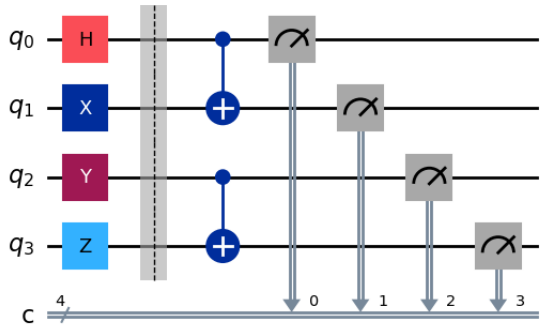


Figure 6.7: Insufficient Depth per Qubit

it, and hence ultimately limits the complexity of quantum states and algorithms that can be trained on it, leading to poor performance. This also increases error rates, however, which makes the circuit vulnerable to quantum errors and noise, reducing the accuracy and data integrity. Often, there aren't enough qubits or nondeterministic operations per second so the circuit can properly benefit from quantum parallelism, which results in a longer run time and reduces the effective performance.

Metric Requirements: The IDQ code smell is quantified using Equation 6.8.

$$\text{Gate Depth per Qubit} = \frac{\text{Total Gate Depth}}{\text{Total Number of Qubits}} \tag{6.8}$$

In the example shown in Figure 6.7, the total gate depth is 3 layers and the number of qubits is 4, leading to an average gate depth of:

$$\text{Average Gate Depth} = \frac{3}{4} = 0.75$$

A value less than 1 for this metric indicates insufficient depth per qubit, highlighting potential inefficiencies and areas for improvement. This IDQ metric suggests that the circuit might not be utilising the full computational potential of each qubit, potentially leading to under-optimised quantum operations.

6.3.5 Selected Machine Learning Classifiers

Once the metric values were validated, they assisted in structuring the dataset into smelly and non-smelly categories. Table 6.3 describes the multi-label dataset constructed, with each file labelled to indicate the presence of quantum-specific code smells. During this process, some degree of class imbalance was observed across different smell categories, which could potentially influence the learning process. To address this, appropriate strategies were considered to ensure that the models did not become biased toward the

Table 6.3: Description of Instances for Each Code Smell in Quantum Dataset

Code smell	Independent Variables	No. of Instances		
		Smelly	Non Smelly	Total
LC	Circuit Width, Depth	583	3519	4102
IB	q_bits, cl_bits	1871	2231	4102
UC	Entanglement, circuit connectivity	1292	2810	4102
EM	measurement count, q_bits, cl_bits	434	3668	4102
LQV	Quantum volume	3242	860	4102
IDQ	Average gate depth per qubit	1944	2158	4102

Table 6.4: Base ML Classifiers Selected in this Study

Base Classifier	Acronym	Ref.
Decision Tree (CART Algo)	DT	[232]
Linear SVM	LinSVM	[87]
Polynomial SVM	PolySVM	[233]
Sigmoid SVM	SigSVM	[234]
Radial Basis Function SVM	RBFSVM	[235]
Bernoulli Naive Bayes	BNB	[236]
Gaussian Naive Bayes	GNB	[237]
Multinomial Naive Bayes	MNB	[238]
Logistic Regression	LR	[79]
Multilayer Perceptron	MLP	[239]
Stochastic Gradient Descent	SGD	[240]
Random Forest	RF	[241]
K-Nearest Neighbor	KNN	[90]
Linear Discriminant Analysis	LDA	[242]

majority classes, thereby maintaining fairness and generalisation in smell detection. This labeled dataset was used to train 14 ML classifiers listed in Table 6.4, a detailed explanation of the same has been mentioned in section 2.5.1.

The models were trained, and their performance was evaluated using accuracy, precision, recall, and F1-score. Accuracy measured the overall correctness, precision indicated the proportion of true positives among predicted positives, recall assessed the model's ability to identify relevant instances, and F1-score provided a balance between precision and recall. This study identifies the models that perform well for the detection of code smells in quantum circuits, which is essential to constructing tools to assist in preserving and enhancing the quality of quantum software.

6.3.6 Statistical Tests

In this study, the Wilcoxon signed-rank test is used to compare the performance differences of 14 selected classifiers to perform the prediction on a curated quantum code smell dataset with six types of smell. This non-parametric test is suitable because the data does not follow a normal distribution, as shown by the Shapiro Wilk [243, 244] test.

It is most useful as a tool for comparing samples of paired data or at least those generated by the same measurement procedure, which allows accuracies to be assigned to measured classifier performance, e.g., as a cross-validated area under the ROC curve (AUC). According to Demsar [245], the magnitudes of the differences (not only ranks) offer more detailed comparisons, and the Wilcoxon signed-rank test is recommended for comparing several classifiers among multiple datasets since it is robust and makes fewer assumptions regarding the distribution of data.

The Wilcoxon Signed-Rank Test requires two hypotheses to be formulated: the Null Hypothesis (H_0) and the Alternative Hypothesis (H_1). The null hypothesis is that the two ML models being compared have equivalent accuracy distributions. In terms of a symbolic representation, this can be written as:

$$H_0: \quad \text{There is no difference in the accuracy distributions of the two classifiers.}$$

The alternative hypothesis (H_1) is that there is a significant difference in the accuracy of the two conditions. It can be written as:

$$H_1: \quad \text{There is a significant difference in the accuracy distribution of the two classifiers.}$$

From there, the test calculates a test statistic based on the sum of the ranks of the differences between observations paired with each other. This statistic is then compared to a critical value or evaluated to obtain a p-value. If the p-value obtained from the test is less than a predetermined significance level ($\alpha = 0.05$), the null hypothesis is rejected. This rejection indicates that there is a statistically significant difference in accuracy between the classifiers.

Mathematically, the test statistic for the Wilcoxon signed-rank test can be represented as:

$$W = \sum_{i=1}^{N} R_i$$

Where W is the test statistic, N is the number of paired observations, and R_i is the

rank assigned to the ith absolute difference between paired observations.

Conducting hypothesis tests with multiple comparisons over the same dataset increases the likelihood of Type I error (i.e., rejecting the null hypothesis when it should be accepted). To mitigate this, the Bonferroni correction procedure was employed. This correction aims to reduce Type I error by adjusting the p-value using the formula:

$$\text{Adjusted } p\text{-value} = \frac{\alpha}{T}$$

In this context, α denotes the initial p-value, while T represents the total number of comparisons conducted.

6.4 Running Example: Bernstein-Vazirani Circuit

To grasp these metrics, consider a quantum circuit featuring *Bernstein-Vazirani*, as depicted in Figure 6.8. The figure shows the Python code and the generated circuit from this code. The Bernstein-Vazirani algorithm is a quantum algorithm that efficiently solves the problem of determining a hidden bit string encoded within a black-box function. Specifically, given a function

$$f(x) = a \cdot x$$

(where $\cdot$ denotes the bitwise dot product), the algorithm aims to find the hidden bit string a. Utilising quantum parallelism, the algorithm prepares a superposition of all possible inputs, applies the oracle function, which encodes the hidden string into the phase of the quantum state, and then uses a second set of Hadamard gates to transform this phase information into measurable output. The final measurement of the qubits reveals the hidden string a in a single query, showcasing a significant improvement over the classical approach, which requires multiple queries. This makes the Bernstein-Vazirani algorithm a powerful example of how QC can solve specific problems more efficiently than classical computing. The original Bernstein-Vazirani circuit from the Qiskit documentation is used. This circuit undergoes analysis to derive both structural and semantic metric values. The extracted values are detailed in Table 6.5 specifying the values of structural and semantic metrics for the given example circuit.

The circuit is evaluated against the conditions for each code smell listed in Table 6.6. The table outlines the metric conditions necessary for identifying the presence of a code smell. Alongside these conditions, the table presents the corresponding values for the Bernstein-Vazirani circuit. These values indicate whether the specific code smell is present in the example circuit or not. Based on the conditions evaluated, the Bernstein-Vazirani circuit contains the **IB** code smell.

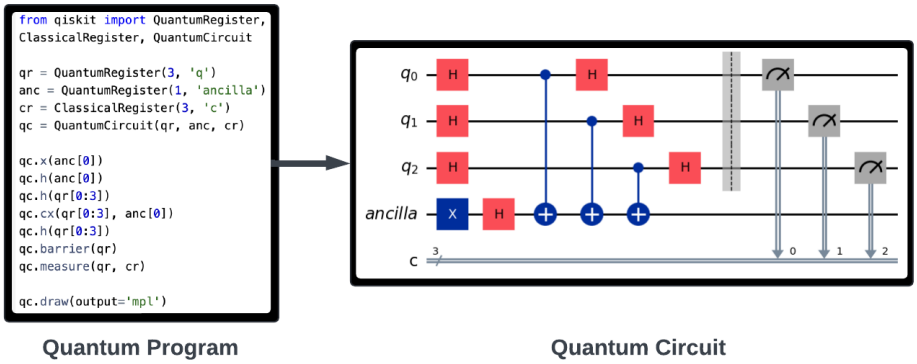


Figure 6.8: Example Bernstein Vazirani Quantum Circuit to show metric values

Table 6.5: Calculation of Quantum Metrics for Bernstein-Vazirani Circuit

Metric	Value
Depth	7
Number of Quantum Bits	4
Number of Classical Bits	3
Number of Measurements	3
Total Gate Operations	12
Gate Counts	{'h': 7, 'cx': 3, 'x': 1}
Circuit Width	7
Average Gate Depth per Qubit	1.75
Entangling Gates	3
Circuit Connectivity	1
Quantum Volume	128
Code Comprehension	7.526

6.5 Results Specific to RQs

This section will address and evaluate the study based on the defined research questions.

6.5.1 Setup Requirements

All experiments are conducted on a workstation with a 2.6 GHz Intel Core i7 13th gen CPU and 16 GB of main memory running 64-bit Windows 11. The Python version is 3.9, and the TensorFlow version is 2.16.1.

Table 6.6: Metric Conditions and Computed Values for Code Smell Detection in Bernstein Vazirani Circuit

Code Smell	Condition	Values
Long Circuit (LC)	$\frac{\text{circuit_width}}{\text{depth}} < 1$	1
Excessive Measurements (EM)	$\frac{\text{num_quantum_bits}+\text{num_classical_bits}}{2\times\text{num_measurements}} < 1$	≈ 1.167
Insufficient Depth per Qubit (IDQ)	$\text{avg_gate_depth_per_qubit} < 1$	1.75
Inconsistent Bits (IB)	$\text{num_quantum_bits} \neq \text{num_classical_bits}$	$4 \neq 3$
Unconnected Circuit (UC)	Circuit Connectivity ≤ 1 and Entangling Gates $= 0$	False
Low Quantum Volume (LQV)	Quantum Volume < 10 and code_comprehension < 1	128

6.5.2 RQ1: What are the researchers' views on quantum-specific code smells?

To validate the defined code smells, an online survey was distributed to 400 reviewers. Out of which 173 filled responses were received. This group comprised industry specialists and academic authors. The industry specialists were further divided into two categories based on their experience: those with less than 10 years of experience and those with more than 10 years of experience. The academic authors were classified into three categories: assistant professors, associate professors, and professors.

Each reviewer was given a detailed briefing about the quantum code smells developed in this study, along with the dataset. They were asked various questions related to the validity and effectiveness of the code smells and to rank these smells based on their severity. Table 6.7 lists and describes each question posed by the reviewers, along with its corresponding scale range. The average ratings from reviewers in each category are detailed in Table 6.8. The reviewers were asked to rank the code smells based on validity and effectiveness using a 5-point scale, where a lower rank indicates a higher severity of the code smell (i.e., 1 denotes the most severe). According to the table, LC is identified as the most severe code smell, with an average rank of 2.7. This is followed by EM, which is regarded as a highly disturbing code smell, with a rank of 3.1. Additionally, they were asked if they agreed that such code smells are present in QPs. The results of their responses are shown in Table 6.9. On average, 73.17% of the reviewers agreed that the proposed code smells affect the functioning of QPs.

Most of the reviewers with experience in quantum programming reviewed and tested the presence of these code smells in a real-world dataset. They provided comments on

Table 6.7: Code Smells Validation Questionnaire

Q. No.	Category	Question	Scale	Description
1	**Validity**	Do you find this code smell relevant and applicable to real-world scenarios?	1: Highly Relevant 5: Not Relevant	Evaluates whether the code smell is commonly encountered and applicable to real-world projects.
2		Is this code smell clearly defined and distinct from other code quality issues?	1: Very Clear 5: Not Clear	Assesses the clarity of the code smell definition and whether it's easy to distinguish from other issues.
3		Can you verify that this code smell exists based on your experience in software development or maintenance?	1: Strongly Agree 5: Strongly Disagree	Checks if the reviewer has personally experienced or observed the code smell in practice.
4		Do you agree that this code smell represents an actual flaw or inefficiency in the codebase?	1: Strongly Agree 5: Strongly Disagree	Confirms if the reviewer agrees that the code smell negatively impacts code quality or efficiency.
5	**Effectiveness**	How effective do you think detecting and addressing this code smell will be in improving code quality?	1: Very Effective 5: Not Effective	Measures the perceived impact of detecting and fixing the code smell on improving the overall quality of the code.
6		Do you believe that removing this code smell will have a noticeable impact on performance/maintainability?	1: Very Impactful 5: Not Impactful	Evaluates how much the removal of the code smell will affect software performance or long-term maintainability.
7		Are the suggested mitigation strategies for this code smell practical and feasible to implement?	1: Very Feasible 5: Not Feasible	Assesses the practicality of the proposed solutions for addressing the code smell.
8		Do you think current tools and techniques are effective in identifying and resolving this code smell?	1: Highly Effective 5: Not Effective	Checks if existing tools or techniques can reliably detect and help address the code smell.
9	**Severity**	How would you rank the severity of this code smell compared to other common code smells?	High / Low	Evaluates how serious this code smell is compared to others in terms of impact on the project.
10		Based on your experience, how frequently do you encounter this code smell, and how impactful is it?	High / Low	Assesses how frequent and impactful the code smell is in practice based on the reviewer's experience.

Table 6.8: Individual Reviewer Ratings for Validity and Effectiveness of Code Smells

Reviewer	**Aspect**	**Code Smell**					
		LC	**UC**	**IB**	**EM**	**LQV**	**IDQ**
Category 1	Validity	2.0	3.7	4.5	4.0	4.2	5.0
(Developers>=10 yrs experience)	Effectiveness	2.6	3.8	4.7	3.2	4.8	4.5
	Severity	High	Low	High	High	Low	High
Category 2	Validity	2.5	4.0	4.3	3.0	3.9	4.6
(Developers<10 yrs experience)	Effectiveness	2.8	4.1	4.6	3.1	3.7	4.7
	Severity	High	High	High	High	High	High
Category 3	Validity	3.0	3.5	4.6	3.5	4.0	4.4
(Professor)	Effectiveness	2.4	4.3	4.5	3.4	4.0	4.6
	Severity	High	High	Low	High	Low	High
Category 4	Validity	2.8	4.2	4.8	2.8	3.6	4.8
(Associate Professor)	Effectiveness	2.9	4.0	4.8	2.9	3.5	4.8
	Severity	High	Low	High	High	Low	High
Category 5	Validity	3.5	3.6	4.1	2.0	3.9	4.6
(Assistant Professor)	Effectiveness	2.7	4.2	3.7	3.8	4.2	4.5
	Severity	High	Low	Low	Low	Low	Low
	Average	2.70	3.75	4.25	3.1	3.90	4.50
	Median	2.8	4.0	4.5	3.1	3.9	4.6

both the presence and impact of these code smells. Some reviewers, particularly industry specialists, acknowledged the presence of these smells but felt they are an inherent part of programming. Additionally, many reviewers highlighted their potential to hinder code maintainability and readability, noting that while these smells are challenging, addressing them could lead to significant improvements in code quality.

One interpretation is that participants perceived the LC smell as particularly harmful due to the significant overhead introduced by long circuits in quantum processing. As one participant remarked, "A long circuit leads to higher error rates and increased decoherence, making it unsustainable for complex operations." In contrast, regarding the EM smell, one participant observed, "While excessive measurements are sometimes necessary, overusing them adds to execution time and affects the reliability of the results better to limit them when possible." As for the IB smell, some participants mentioned, "Branching isn't inherently problematic, but unnecessary branching in quantum circuits can lead to inefficiency, especially if better alternatives are available." Interestingly, 40% of participants found the LQV smell harder to identify, possibly due to the difficulty in recognising poorly defined variables within quantum circuits. Meanwhile, for UC and IDQ, the consensus was clear: "Both unclear qubit assignments and improperly defined quantum gates contribute to circuit errors, but they're secondary concerns compared to

Table 6.9: Survey results per smell

	LC	UC	LQV	EM	IDQ	IB	Average
Agree	78.00%	73.00%	70.00%	66.00%	64.00%	88.00%	73.17%
Disagree	12.00%	17.00%	20.00%	27.00%	21.00%	7.00%	17.33%
Don't know	10.00%	10.00%	10.00%	7.00%	15.00%	5.00%	9.50%

long circuits."

While addressing these code smells presents challenges, especially with long circuits, doing so could significantly enhance the performance and maintainability of quantum programs.

Table 6.10: Descriptive Statistics for Quantum Metrics

Quantum Metrics	mean	std	min	25%	50%	75%	max
depth	3.852	6.749	0.0	1.0	2.0	4.0	110.0
num_quantum_bits	7.748	113.833	1.0	2.0	3.0	6.0	4200.0
num_classical_bits	6.204	113.842	0.0	0.0	2.0	4.0	4200.0
num_measurements	1.677	8.759	0.0	0.0	0.0	2.0	367.0
num_gates	5.995	17.961	0.0	1.0	2.0	8.0	736.0
circuit_width	13.952	227.641	1.0	3.0	4.0	10.0	8400.0
avg_gate_depth_per_qubit	1.157	1.596	0.0	0.253	1.0	1.5	47.0
code_comprehension	2.145	3.158	0.0	0.0	1.0	2.672	62.0
Entanglement (Entangling Gates)	4.651	131.382	0.0	0.0	1.0	3.0	8400.0
Quantum Volume	3.170	2.030	0.048	0.979	2.0	8.0	1.300

Summary of RQ1: The study conducted a comprehensive review of code smells in QPs, involving 173 industry specialists and academic authors. Reviewers briefed on the identified code smells, ranked them based on severity, and assessed their validity and effectiveness. On average, 73.17% of reviewers agreed that these code smells impact QP functionality. Notably, LC was identified as the most severe, followed by EM. While some acknowledged the presence of these code smells, others viewed them as intrinsic to programming.

6.5.3 RQ2: How does metrics analysis contribute to a more comprehensive analysis of code smells in quantum programming?

This study incorporated the use of both structural and semantic metrics. When proposing new metrics, it is essential to provide detailed information about them. Table 6.10

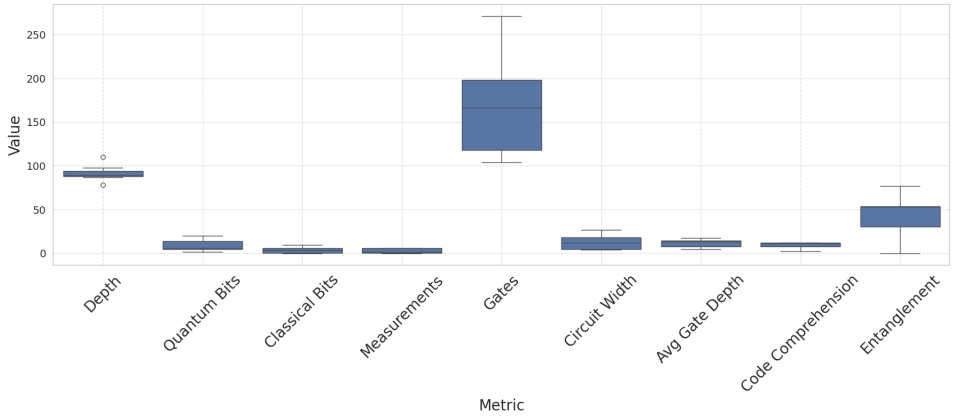


Figure 6.9: Outliers in Quantum Metrics

describes the descriptive statistics of all the metrics used in the study. This table presents various statistical measures for each metric, including the mean, median, standard deviation, minimum, and maximum values. By providing these statistical ranges and measures, the table helps to understand the distribution, central tendency, and variability of the metrics, thereby giving a comprehensive overview of their characteristics and behavior within the dataset.

Another important aspect of these metrics is the presence of outliers in the dataset, as shown in Figure 6.9. This figure shows whether the metrics have any values outside their usual range. For example, the 'depth' metric has 2 outliers, whereas the 'gates' metric falls between 100 and 250. This visualization helps understand the distribution and differences in the metrics dataset.

Additionally, the correlation analysis provides more intricate information on the interrelationship of different metrics as shown in Table 6.11. The correlation coefficients allow researchers to understand when one metric changes and how the other metrics would change. If two metrics are strongly positively correlated, then they tend to grow together in proportion. For QP, when the 'Total Gate Operations' is highly positively correlated to the depth of quantum circuits, it indicates the case that as the depth gets larger, using more gates. This helps in understanding the quantum circuits and components.

A negative correlation, however, means that as one metric grows, the other is likely to decrease or stay constant. For example, an inverse correlation between 'depth' and 'q_bits' would show that increasing the depth might lower or sustain the number of qubits consumed. These insights can power a streamlining of quantum programming, thereby realising more effective lines of code for developers.

Additionally, great effort in predicting weak or null relationships between metrics allows for changes in one metric to be made without driving the other to a significant extent. This insight is necessary to better inform where optimisations should be done and

Table 6.11: Correlation Analysis Results of Metrics

	depth	q_bits	cl_bits	measurements	gates	circuit_width	avg_gate_depth	code_comprehension	Entanglement	Quantum Volume
depth	1.0									
q_bits	-0.0055	1.0								
cl_bits	-0.0083	0.9994	1.0							
measurements	0.0753	0.0680	0.0700	1.0						
gates	0.6595	0.0372	0.0351	0.5452	1.0					
circuit_width	-0.0069	0.9999	0.9999	0.0690	0.0362	1.0				
avg_gate_depth	0.6607	-0.0281	-0.0243	-0.0150	0.3202	-0.0262	1.0			
code_comprehension	0.5736	-0.0038	-0.0032	0.0845	0.4086	-0.0035	0.2303	1.0		
Entanglement	0.0095	0.5767	0.5766	0.0270	0.0106	0.5767	-0.0036	-0.0079	1.0	
Quantum Volume	0.2457	0.0013	0.0005	-0.0012	0.2305	0.0009	0.0520	0.0396	-0.0006	1.0

Table 6.12: Performance of Different Feature Sets for Code Smell Detection

Features	**LC**				**UC**				**LQV**			
	Acc	**F1**	**Recall**	**Precision**	**Acc**	**F1**	**Recall**	**Precision**	**Acc**	**F1**	**Recall**	**Precision**
With semantic features	75.00%	73.10%	74.00%	76.00%	70.00%	85.62%	71.00%	72.00%	98.00%	97.00%	97.00%	96.00%
With structural features	80.00%	79.00%	81.00%	79.00%	75.00%	84.69%	76.00%	83.07%	98.00%	97.00%	97.87%	97.00%
Both	**99.88%**	**99.88%**	**99.63%**	**96.00%**	**99.63%**	**96.95%**	**97.19%**	**97.28%**	**99.88%**	**98.00%**	**99.00%**	**97.00%**
Features	**EM**				**IDQ**				**IB**			
	Acc	**F1**	**Recall**	**Precision**	**Acc**	**F1**	**Recall**	**Precision**	**Acc**	**F1**	**Recall**	**Precision**
With semantic features	85.62%	68.00%	71.00%	72.00%	75.90%	76.54%	66.00%	64.00%	75.00%	73.00%	76.00%	74.00%
With structural features	85.62%	90.50%	76.00%	74.00%	70.00%	90.50%	71.00%	69.00%	80.00%	78.00%	81.00%	79.00%
Both	**99.63%**	**97.00%**	**99.00%**	**98.79%**	**98.27%**	**96.95%**	**97.28%**	**97.28%**	**99.63%**	**98.05%**	**97.19%**	**98.79%**

in what parts of QP have the most impact on performance.

Summary of RQ2: From this analysis, it became evident that there were little or no outliers in the dataset, implying that overall, the data was consistent. The statistical description provided detailed insights into the range of values for each metric, enhancing the understanding of their distribution and variability. The correlation analysis revealed both positive and negative relationships between metrics, demonstrating intricate patterns within the dataset. Positive correlations represented co-dependencies between structural elements, and negative correlations indicated potential strategies to optimise.

6.5.4 RQ3: Does a model trained on a dataset with both structural and semantic features perform better than one trained on only structural or semantic metrics?

This study investigated the impact of utilising different feature sets on model performance for quantum code smell detection. Specifically, three feature sets were evaluated: one with only semantic features, one with only structural features, and one combining both types of features. The semantic features capture the meaning and context of the code, while structural features represent the code's architecture and organisation.

Table 6.12 presents the experimental results of the performance of different feature sets. For LC code smell, using both features results in a 20% increase in precision and

Table 6.13: Performance of Various Machine Learning Models on Different Code Smells

Model	LC				UC				IB			
	Acc	Prec	Rec	F1	Acc	Prec	Rec	F1	Acc	Prec	Rec	F1
LR	0.855	0.7311	0.855	0.7882	0.5274	0.7508	0.5274	0.3642	0.5274	0.7508	0.5274	0.3642
MLP	0.9829	0.9827	0.9829	0.9827	0.9695	0.9696	0.9695	0.9695	0.9695	0.9696	0.9695	0.9695
SGD	0.1449	0.876	0.1449	0.0366	0.4726	0.7508	0.4726	0.3033	0.4726	0.7508	0.4726	0.3033
RF	1.0000	1.0000	1.0000	1.0000	0.9866	0.9868	0.9866	0.9866	0.9903	0.9903	0.9903	0.9903
KNN	0.9866	0.9867	0.9866	0.9866	0.9415	0.9423	0.9415	0.9414	0.9415	0.9423	0.9415	0.9414
LDA	0.8855	0.8759	0.8855	0.8789	0.8307	0.8472	0.8307	0.8298	0.8307	0.8472	0.8307	0.8298
DT	0.9805	0.9813	0.9823	0.9292	0.9476	0.9778	0.9098	0.9426	0.9409	0.9661	0.9048	0.9344
LinSVM	0.7941	0.3214	0.3781	0.3474	0.6322	0.6229	0.5619	0.5908	0.6322	0.6229	0.5619	0.5908
PolySVM	0.8562	1.0000	0.0084	0.0167	0.5286	1.0000	0.0026	0.0051	0.5286	1.0000	0.0026	0.0051
SigSVM	0.8562	1.0000	0.0084	0.0167	0.5274	1.0000	0.0000	0.0000	0.5274	1.0000	0.0000	0.0000
RBFSVM	0.8562	1.0000	0.0084	0.0167	0.5274	0.0000	0.0000	0.0000	0.5274	0.0000	0.0000	0.0000
GNB	0.8562	0.8769	0.8562	0.7911	0.5286	0.7511	0.5286	0.3669	0.5286	0.7511	0.5286	0.3669
MNB	0.8611	0.8805	0.8611	0.8025	0.5274	0.2782	0.5274	0.3642	0.5274	0.2782	0.5274	0.3642
BNB	0.9719	0.9728	0.9719	0.9707	0.9635	0.9635	0.9635	0.9635	0.9635	0.9635	0.9635	0.9635
Model	**LQV**				**IDQ**				**EM**			
	Acc	Prec	Rec	F1	Acc	Prec	Rec	F1	Acc	Prec	Rec	F1
LR	0.5274	0.7508	0.5274	0.3642	0.5262	0.7507	0.5262	0.3628	0.9050	0.9140	0.9050	0.8599
MLP	0.9695	0.9696	0.9695	0.9695	0.9963	0.9963	0.9963	0.9963	0.9781	0.9791	0.9781	0.9784
SGD	0.4726	0.7508	0.4726	0.3033	0.5262	0.7507	0.5262	0.3628	0.0950	0.9140	0.0950	0.0165
RF	0.9903	0.9903	0.9903	0.9903	0.9988	0.9988	0.9988	0.9988	0.9915	0.9916	0.9915	0.9913
KNN	0.9415	0.9423	0.9415	0.9414	0.9878	0.9879	0.9878	0.9878	0.9440	0.9428	0.9440	0.9433
LDA	0.8307	0.8472	0.8307	0.8298	0.9622	0.9641	0.9622	0.9623	0.9281	0.9339	0.9281	0.9306
DT	0.9409	0.9661	0.9048	0.9344	1.0000	1.0000	1.0000	1.0000	0.9439	1.0000	0.4685	0.6381
LinSVM	0.6322	0.6229	0.5619	0.5908	0.6565	0.6482	0.6015	0.6240	0.8380	0.0339	0.0256	0.0292
PolySVM	0.5286	1.0000	0.0026	0.0051	0.5262	0.0000	0.0000	0.0000	0.9062	1.0000	0.0128	0.0253
SigSVM	0.5274	1.0000	0.0000	0.0000	0.5262	1.0000	0.0000	0.0000	0.9050	1.0000	0.0000	0.0000
RBFSVM	0.5274	0.0000	0.0000	0.0000	0.5262	0.0000	0.0000	0.0000	0.9050	0.0000	0.0000	0.0000
GNB	0.5286	0.7511	0.5286	0.3669	0.4750	0.7510	0.4750	0.3073	0.9062	0.9150	0.9062	0.8628
MNB	0.5274	0.2782	0.5274	0.3642	0.4799	0.7521	0.4799	0.3180	0.9050	0.8190	0.9050	0.8599
BNB	0.9635	0.9635	0.9635	0.9635	0.9805	0.9805	0.9805	0.9805	0.9488	0.9516	0.9488	0.9401

a 26.78% increase in F1-score compared to using only semantic features. Similarly, for the structural features, the dataset shows a 17% and 20.88% reduction in precision and F1-score, respectively, compared to the combined dataset.

For UC code smell detection, precision and F1-score drop by 25.28% and 11.33%, respectively, when using only semantic features, and by 14.21% and 12.26%, respectively, when using only structural features, compared to the combined feature set.

In the case of LQV code smell detection, using both semantic and structural features results in increased F1-score and recall by 1.00% and 2.00%, respectively, compared to using only one type of feature, where using only structural features, the dataset showed 1% and 1.13% reduction in the metrics.

Similar trends are observed for EM, IDQ, and IB code smells. For example, for EM code smell detection, using only semantic features results in a 29.00% reduction in F1-score and a 26.79% reduction in precision, while the structural features-only dataset shows

a decrease of 6.50% and 24.79%, respectively.

These findings indicate that relying solely on either semantic or structural features significantly compromises model performance for quantum code smell detection. The datasets combining both feature types consistently yield higher accuracy across all metrics, highlighting the importance of integrating semantic and structural information for effective code smell detection.

Summary of RQ3: This analysis concluded that leveraging both structural and semantic metric analyses for code smell detection yields superior results compared to using either type of metric alone. Among the different code smells, the LC code smell showed the most significant performance improvement when using both structural and semantic features, with an increase of approximately 24.88% in accuracy, 26.78% in F1-score, 25.63% in recall, and 20% in precision compared to using only semantic features. On the other hand, the EM code smell exhibited the least improvement, with a comparatively lower increase in performance metrics, ranging from 8.13% to 29.00%, when utilising both feature types compared to using only semantic features.

6.5.5 RQ4: How do various ML classifiers contribute to the classification of code smells?

This subsection investigates the detection performance of the 14 individual classifiers in detecting quantum code smells. Table 6.13 presents the result of these ML classifiers based on four evaluation metrics: accuracy, precision, recall, and F1 score.

The table compares the performance of various ML models across different code smells: LC, UC, IB, LQV, IDQ, and EM. Overall, the RF classifier consistently outperforms other models with near-perfect scores of all metrics across almost all code smells. MLP and KNN also show strong performance, particularly for LC and EM, with high metrics close to those of RF. Conversely, models like SGD and various SVM kernels (PolySVM, SigSVM, RBFSVM) perform poorly on several code smells, often with low recall and F1 scores, indicating they are not well-suited for these tasks. LDA and DT classifiers perform relatively well but are less consistent compared to RF, MLP, and KNN. LR struggles with UC and IB but performs adequately on LC and EM. Models like GNB and BNB have variable performance, with BNB being notably better. SVM models with different kernels (Poly, Sigmoid, RBF) often achieve high precision but suffer from extremely low recall, making them unreliable for detecting code smells. In summary, the RF classifier is the best performer for detecting code smells, while models like MLP and KNN are also highly

effective. Conversely, SVM models and SGD show significant limitations, particularly in recall and F1 scores.

In addition to the performance metrics, Figure 6.10 presents boxplots of the accuracy of each ML classifier across different cycles of testing. The figure illustrates that, apart from SGD, many classifiers have performed well, as indicated by their higher accuracy boxes. Specifically, EM and LC display the highest whiskers in the boxplots, suggesting they achieved the highest accuracy levels, while other code smells exhibit more variation in performance. The results indicate that the RF classifier consistently outperforms other models across various code smells, achieving near-perfect scores in accuracy, precision, recall, and F1. MLP and KNN also demonstrate strong performance, particularly for LC and EM code smells. This highlights the capability of machine learning models to learn complex, nonlinear patterns from the metric-based features and generalise well across multiple smell types. In particular, RF's ensemble structure makes it well-suited for capturing threshold-based relationships, while MLP leverages its DL capacity to model intricate decision boundaries. Conversely, models like SGD and various SVM kernels (Poly, Sigmoid, RBF) perform poorly, especially in recall and F1 scores. LR and DT classifiers perform adequately but less consistently. Naive Bayes classifiers (GNB and BNB) have variable performance. Overall, these results emphasise the utility of machine learning techniques in automating code smell detection and reinforce RF and MLP as the most reliable and effective classifiers in this context.

For each feature selection technique, machine learning (ML) classifiers are ranked based on the area under the ROC curve (AUC value). Figure 6.11 highlights the best ML classifiers when the eight classifiers that provide ROC results are compared. The figure illustrates the ranking from best to worst for each code smell, allowing a clear comparison of classifier performance. Across all code smells analysed, the Random Forest (RF) classifier consistently emerges as the best performer, achieving an AUC of 1, which underscores its robustness and reliability in detecting quantum code smells compared to other classifiers. The high AUC values achieved by RF suggest it effectively captures the underlying patterns and relationships in the data, crucial for accurate detection.

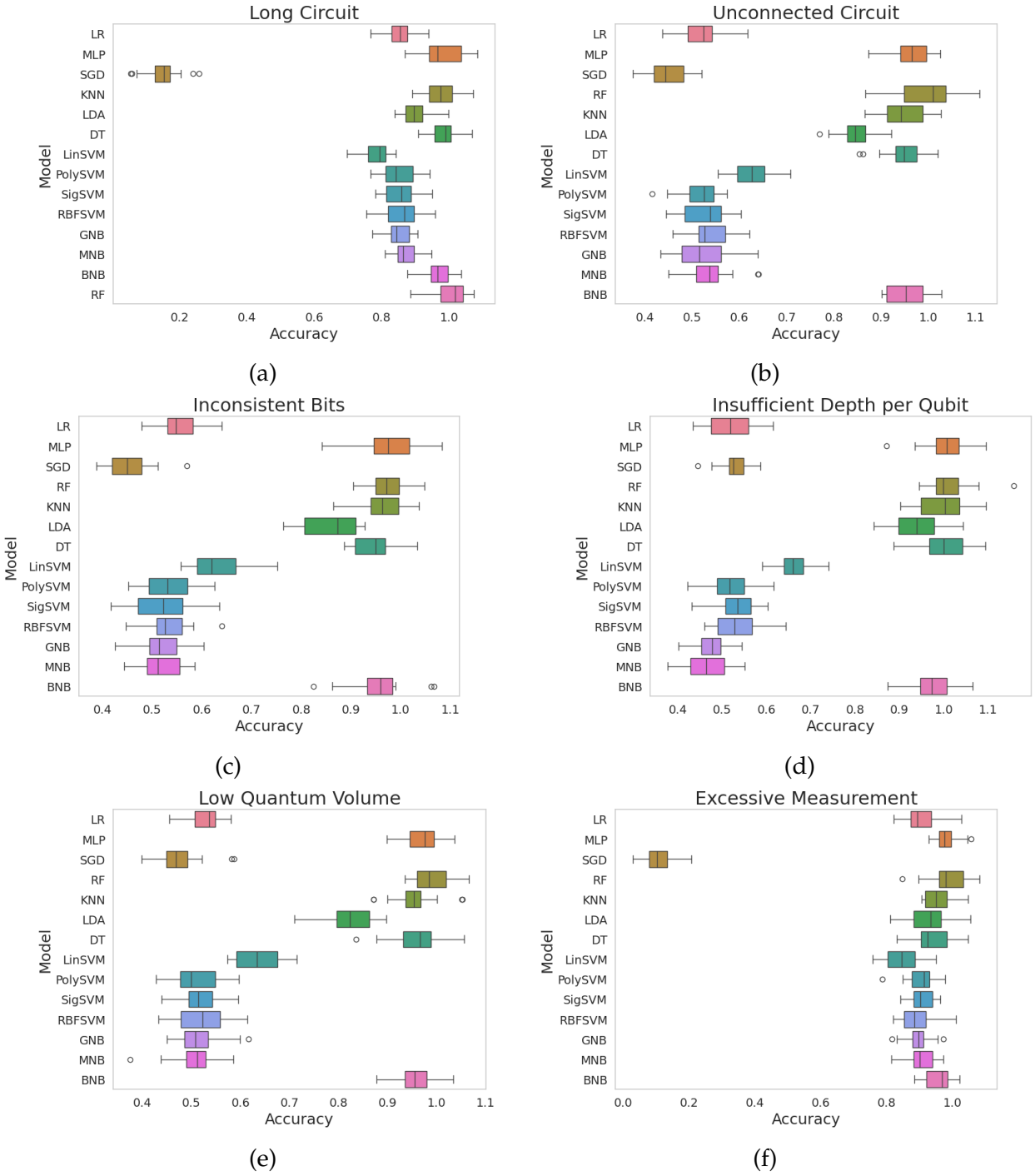


Figure 6.10: Box plots of the performance (accuracy) exhibited by the considered machine learning models for all the six smells.

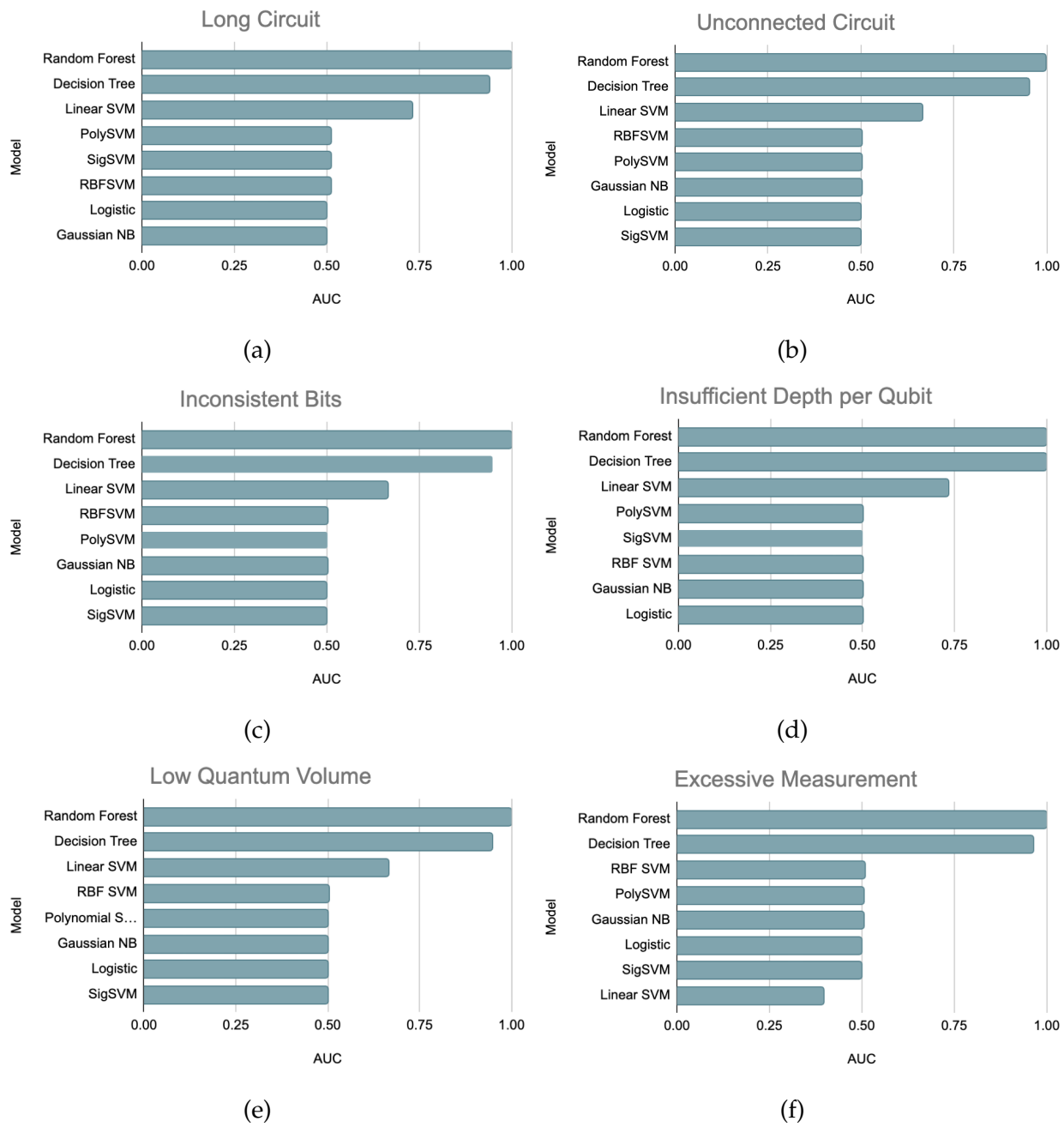


Figure 6.11: ROC AUC of Classifiers from best to worst for different code smells detection

Summary of RQ4: The results indicate that the RF classifier consistently outperforms other models across various code smells, achieving near-perfect scores in accuracy, precision, recall, and F1. MLP and KNN also demonstrate strong performance, particularly for LC and EM code smells. Conversely, models like SGD and various SVM kernels (Poly, Sigmoid, RBF) perform poorly, especially in recall and F1 scores. LR and DT classifiers perform adequately but less consistently. Naive Bayes classifiers (GNB and BNB) have variable performance. RF and MLP are consistently the most reliable and effective classifiers.

Table 6.14: Classifiers statistical performance results Across Different Smell Types

Classifier	LC		UC		IB		LQV		IDQ		EM		Total	
	Wins	Losses	Wins	Losses	Wins	Losses	Wins	Losses	Wins	Losses	Wins	Losses	Wins	Losses
RF	11		13		13		12		13		13		75	0
MLP	11	2	13		13		13		13		13		76	2
SGD	13		13		13		9		6	7	9	4	63	11
BNB	13			13	5	8	8		13		1	12	40	33
KNN	11			13		13	13			13		10	24	49
DT	13		5	8	5	8	5	6	5	8		13	33	43
MNB		13		13		13		13		13		13	0	78
LR	2	11	9	4	4			13	6	7		12	21	47
RBFSVM	1	12	9		13		2	11	6	7	1	12	32	42
SigSVM		13		9		13		13	6	7		13	6	68
LinSVM	9		13		13		13		13		2	11	63	11
LDA		13	13		2	11	13		10	1	3	10	41	35
PolySVM	1	12		13		13		13			13		14	51
GNB	1	6	10		10	2		13		7	1	12	22	40

Table 6.15: Classifiers paired comparisons results

Classifier	RF	MLP	SGD	BNB	KNN	DT	MNB	LR	RBFSVM	SigSVM	LinSVM	LDA	PolySVM	GNB	Win %
RF		2	5	5	8	4	5	8	6	5	4	10	10	3	96.15
MLP				3	6	4	22	5	3	17	2	6	8		97.44
SGD					5	3	20	2	4	19		5	5		80.77
BNB					7	2	2	7	8	5	1		8		51.28
KNN				2		5	2	7	2		1	2	3		30.77
DT				2	2		3	7	5			3	1	10	42.31
MNB															0.00
LR			2	5	1	3	3		2	1	1	2	1		26.92
RBFSVM				2		8	3	2		5	1	3	1	7	41.03
SigSVM			2	1	1			1				1			7.69
LinSVM				4	9	7	8	2	5	9			10	9	80.77
LDA					5	4	7	5	2	4	1		2	11	52.56
PolySVM			2	2	3	2	1	1		2		1			17.95
GNB				7	2	1	2		5	1		2	2		28.21
Loss %	0.00	2.56	14.10	42.31	62.82	55.13	99	60.26	53.85	87.18	14.10	44.87	65.38	51.28	

6.5.6 RQ5: How does statistical analysis validate the dataset's efficacy for accurately detecting code smells?

This study utilised the Wilcoxon statistical test to analyse whether there is a significant difference in the performance between the two classifiers. The null hypothesis for the Wilcoxon test defines that there is no difference in performance between the two classifiers. If the null hypothesis is rejected, it indicates that the performance of classifier A is significantly higher than that of classifier B. In such cases, classifier A is referred to as the "winner" and classifier B as the "loser." Conversely, if the null hypothesis is accepted (i.e., the difference in performance between A and B is insignificant), neither classifier is designated as a winner or a loser.

The total number of wins and losses for each classifier across various code smell datasets is displayed in Table 6.14. The statistical test results align well with the ML classifiers' outcomes. Although MLP was the most successful classifier, with 76 wins, RF stood out as the least losing classifier, with 0 losses. In contrast, MNB was the poorest performer, with 0 wins and 78 losses. This table effectively highlights which models perform better for each code smell.

Expanding on the results in this table, the wins and losses for each pairwise classifier comparison are shown in Table 6.15. For a given row classifier, the row indicates the total number of wins against the column classifier. Similarly, for a given column classifier, the column shows the total number of losses against the row classifier. The last row and column calculate the percentage of total wins and losses, respectively, for each classifier out of all possible paired comparisons. Each classifier has a total of 78 paired comparisons (6 smells multiplied by 13 classifiers). Note that the cumulative sum of wins and losses for a classifier may not equal 78, as the acceptance of the null hypothesis indicates no significant difference in performance, leading to neither classifier being reported as a winner or loser. Notably, MLP has the highest win percentage at 97.44%, while RF has the lowest loss percentage at 0%.

Summary of RQ5: The Wilcoxon statistical test results reveal that the MLP classifier emerged as the top performer with 76 wins, achieving the highest win percentage of 97.44%. Conversely, the RF classifier demonstrated remarkable robustness with the lowest loss percentage at 0%, making it the least-losing classifier. When compared to other classifiers, the MNB classifier performed the worst, with 0 wins and 78 losses, demonstrating its general inefficiency. These results demonstrate how well MLP and RF perform on a variety of code smells, and they also show that MNB is by far the least successful classifier.

Table 6.16: Comparison of state-of-the-art Approaches on Common Code Smells

Study	Ref	Long Circuit (LC)	Excessive Measurements (EM)	Inconsistent Bits (IB)	Method
The Smelly Eight	([70])	99.15%	99%	-	QSmell Tool
Towards Metrics for Quantum Circuits Understandability	([131])	-	-	**99.93%**	Metric analysis
This Study	-	**99.88%**	**99.63%**	99.63%	Structural and semantic metric analysis

6.5.7 RQ6: How does the proposed methodology compare to existing state -of-the-art approaches in detecting quantum code smells?

In recent years, various studies have proposed different methodologies for detecting quantum code smells, each focusing on its perspective and selecting a unique set of code smells. This has led to the proliferation of numerous quantum code smells and associated metrics, some of which are repetitive or overlapping. Moreover, certain code smells identified in previous research are either irrelevant or rarely encountered in real-world applications, making the analysis of such metrics inefficient and a potential drain on programming resources.

To address this issue, this study focuses on identifying the most relevant and frequently occurring quantum code smells from previous research while proposing new, meaningful code smells based on insights from industry experts and reviewers. By narrowing down the scope to only the most impactful and common code smells, this research ensures that efforts are concentrated on detecting those that genuinely affect code quality.

This RQ will investigate how the proposed methodology compares with existing state -of-the-art approaches in detecting the most relevant and common quantum code smells. The comparison will focus on evaluating the performance of different methodologies from prior studies. Table 6.16 presents a comparison with the state -of-the-art approaches. As shown, this study validated the results for three existing code smells, while proposing three additional ones. The results indicate that for two out of the three existing code smells, the proposed methodology outperforms the state -of-the-art approaches. These findings demonstrate the effectiveness of the structural and semantic-based approach in detecting quantum code smells.

Summary of RQ6: The proposed methodology outperformed existing state -of-the-art approaches, achieving an accuracy increase of up to 15% for two out of three validated code smells. By integrating structural and semantic analysis, it effectively identified the most relevant quantum code smells, validating its superiority in accuracy and detection efficiency compared to traditional methods.

6.6 Discussion

This chapter implements structural and semantic metric analysis for code smell detection using a publicly available quantum dataset, LintQ, which consists of 7,568 QPs. This study demonstrates the significance of combining structural and semantic analysis to improve how code snippets are reviewed for metric generation. To reach this goal, this study presents a new proposal that combines semantic analysis, i.e., the analysis of the meaning of the model and structural metrics, with the detection of quantum code smells. This holistic approach combines both structural and semantic metrics to give an overall measurement of the quality of the code. These numbers are used as thresholds to detect six common quantum code smells in QPs, which were validated during the work.

As a result of this thorough analysis, a labelled dataset comprising 12 structural and semantic metrics, as well as labels for various code smells, was developed. In total, the dataset consists of 4,102 real-world QPs. After analysing the structural and semantic metrics, the results show that combining both structural and semantic metrics for code smell detection helps to increase the detection accuracy compared to the case of using only structural or semantic metrics separately.

Industry and academic experts reviewed the six code smells (LC, UC, IDQ, EM, IB, and LQV), evaluating their significance and implications. These reviews not only validated this analysis but also provided valuable insights into the practical relevance of these code smells for developers. The study plans to use the methodology of structural and semantic analysis in other datasets for future work. Furthermore, the reviewers made the point that other code smells should be considered responsible, and this opens up a big opportunity for future research. Such future exploration could have the potential to provide a deeper insight into the quality of quantum code and possibly motivate the development of more sophisticated code smell detection tools.

Chapter 7

QNet: Exploring Deep Learning for Quantum Code Smell Detection

This chapter introduces **QNET**, a DL-based framework designed to detect QCS effectively. By leveraging models such as CNN, GRU, and LSTM, along with their hybrid combinations, QNET explores the complementary strengths of these architectures for quantum smell detection. Using a curated dataset of 136 open-source QP projects, the study evaluates multiple configurations and compares them against transfer learning baselines like BERT. Results demonstrate that the hybrid DL model achieves superior accuracy, highlighting the promise of ensemble-style DL in advancing automated quality assessment for quantum software. This chapter is organized as follows: Section 7.1 gives the introduction of this chapter. Section 7.2 outlines the proposed methodology, presenting a detailed account of the steps undertaken in this study. Section 7.3 discusses the results. Finally, Section 7.4 summarizes the chapter. The work presented in this chapter has been published in Software Quality Journal [246].

7.1 Introduction

The availability of dedicated quantum programming languages and frameworks, such as Cirq by Google, Q# by Microsoft, and Qiskit by IBM, has been instrumental in accelerating the development of more complex QPs [222]. These tools offer an abstraction layer that simplifies the implementation of quantum algorithms and allows practitioners to focus on the core logic of their applications. With the rise of Quantum-as-a-Service (QaaS) platforms providing access to universal quantum devices, QP has become more accessible to a broader community of researchers and developers [223].

The continuous availability of QPs is crucial for the seamless functioning of quantum

applications. However, constant monitoring and maintenance have become challenging as QC technology advances due to several factors [224, 63].

In classical software development, it has been observed that poor design and implementation choices [171, 179], often stemming from developers' lack of experience and skills, can hinder effective maintenance [221, 226]. Similarly, as the field of QC is emerging, programming practices are still evolving rapidly, and related libraries and frameworks frequently undergo significant revisions [225]. Consequently, QPs often consist of code pieced together to accommodate these changes. This code-patching approach can lead to potential issues and complexities as QPs grow in size and complexity [227].

Recognizing these challenges, it is crucial for the software engineering research community to proactively investigate techniques to aid in the maintenance of QPs before widespread problems arise [21, 226]. Despite the growing importance of QC, the software engineering research community still needs to fully explore the maintenance of QPs from a purely quantum perspective.

In the realm of software engineering, prior research has extensively explored maintenance issues, mainly focusing on indicators known as "code smells", and methodologies to identify and measure their impact on classical software [228, 127, 229]. Quantum code smell (QCS) detection in the context of QC is an emerging and relatively unexplored area of research. One such noteworthy contribution in this area is the work by Chen et al. [70], where they introduced a tool designed for detecting QCS in QPs. Additionally, Openja et al. in [68] employed machine learning algorithms for a similar purpose. Nevertheless, only a few studies have been dedicated to identifying QCS in QPs.

Considering the gaps mentioned above, this study investigates the following research questions (RQs):

RQ1: How do QCS influence the frequency of errors in quantum computers?

This RQ explores the complex relationship between QCS and the errors they amplify within quantum computers. It seeks to understand how QCS affects the reliability and performance of quantum systems.

RQ2: What is the relative impact of different QCS on the performance of quantum systems?

Each QCS carries a severity value that indicates its potential to escalate error likelihood. This RQ aims to rank QCS by analyzing their severity values and provides insights into identifying QCS that pose higher risks of error amplification compared to those with lesser consequences.

RQ3: Do the outcomes of DL models, developed with distinct hyper-parameters, exhibit complementary behaviour?

This RQ focuses on conducting an in-depth analysis of the hyper-parameters in CNN, GRU, and LSTM networks that can assist in selecting the ideal layers for constructing a hybrid model. This RQ explores whether the chosen layers and hyper-parameters for each specific model align effectively with their respective capabilities in detecting various QCS.

RQ4: What are the predictive capabilities of different DL models developed for QCS detection?

This RQ evaluates diverse DL models across various performance metrics to assess their effectiveness in detecting QCS. Comparison among the Receiver Operating Characteristic Curve (ROCC) of different models employed for QCS detection is featured to categorize and quantify the results. Each model is assessed independently and individually based on these specified performance parameters.

RQ5: What is the effectiveness of the proposed model when compared with well-known DL models and pre-trained transfer learning model?

This RQ assesses the predictive power of the proposed model by comparing it to various models. Additionally, it contrasts hybrid models with transfer learning, using a pre-trained BERT model trained on classical computers code smell detection for QCS detection. The study aims to determine if the proposed model surpasses transfer learning, offering insights into both approaches' strengths and weaknesses in QCS detection.

Thus, the primary contributions of this research study are threefold. Firstly, it investigates the domain of QCS and its potential impact on errors within QPs, recognizing the significance of addressing these errors to enhance quantum computer performance. Secondly, it explores the feasibility of employing DL techniques to detect QCSs within the code, aiming to improve code quality. Lastly, the study delves into the potential of hybrid DL algorithms, merging multiple DL techniques to ascertain whether they can surpass current methods and elevate QCS detection to new heights.

To the authors' knowledge, this study is the first to apply DL models to detect QCS. The study's results confirm the superiority of the proposed model as it outperforms the individual DL and pre-trained model. Furthermore, the proposed model was found to be

comparable with baseline techniques. Software practitioners can use this framework to detect QCS. Therefore, focusing on these smells and their early detection would guarantee a maintainable and better-quality product.

7.2 Experimental Framework

This section concisely explains the systematic approach employed for detecting QCS. The overview encompasses a sequence of essential steps undertaken to achieve this objective.

7.2.1 Overview

The initial phase involved the collection of 136 QPs from GitHub repositories. These programs were primarily built using the Qiskit libraries and were authored in the Python programming language. The next step included separating quantum circuits from the codes to create execution matrices from quantum codes.

Once the execution matrices were established, the QSmell detection tool was engaged. Chen et al. [70] developed this tool to identify and detect eight distinct categories of QCS in QPs. This tool harnessed the matrices to determine the presence of QCS within the quantum circuits. This analysis created a manually annotated dataset, where code was categorised as either smelly or non-smelly. Further, DL techniques are utilised by conducting an ablation study to determine the optimal hyperparameters and layers for the model, and a hybrid model is proposed. A detailed explanation of all the steps is in the subsequent subsections.

7.2.2 Dataset Preparation

In the rapidly evolving landscape of technology, QC stands as a burgeoning field with limited project availability. To conduct this research effectively, high-quality quantum projects containing codes focused on constructing functional quantum circuits for real-world quantum computers were searched. Github repositories were explored, particularly those housing Qiskit, a prominent open-source software development kit (SDK) for QC.

Since quantum code is predominantly scripted in Python, the focus naturally turned toward GitHub repositories hosting Python-centric content. After conducting a comprehensive manual search of around 200 GitHub projects, the process involved eliminating duplicate and irrelevant projects, resulting in a curated dataset of 136 relevant projects for this study. Among these projects were essential components of quantum code related to significant gate operations and quantum measures, laying the groundwork for

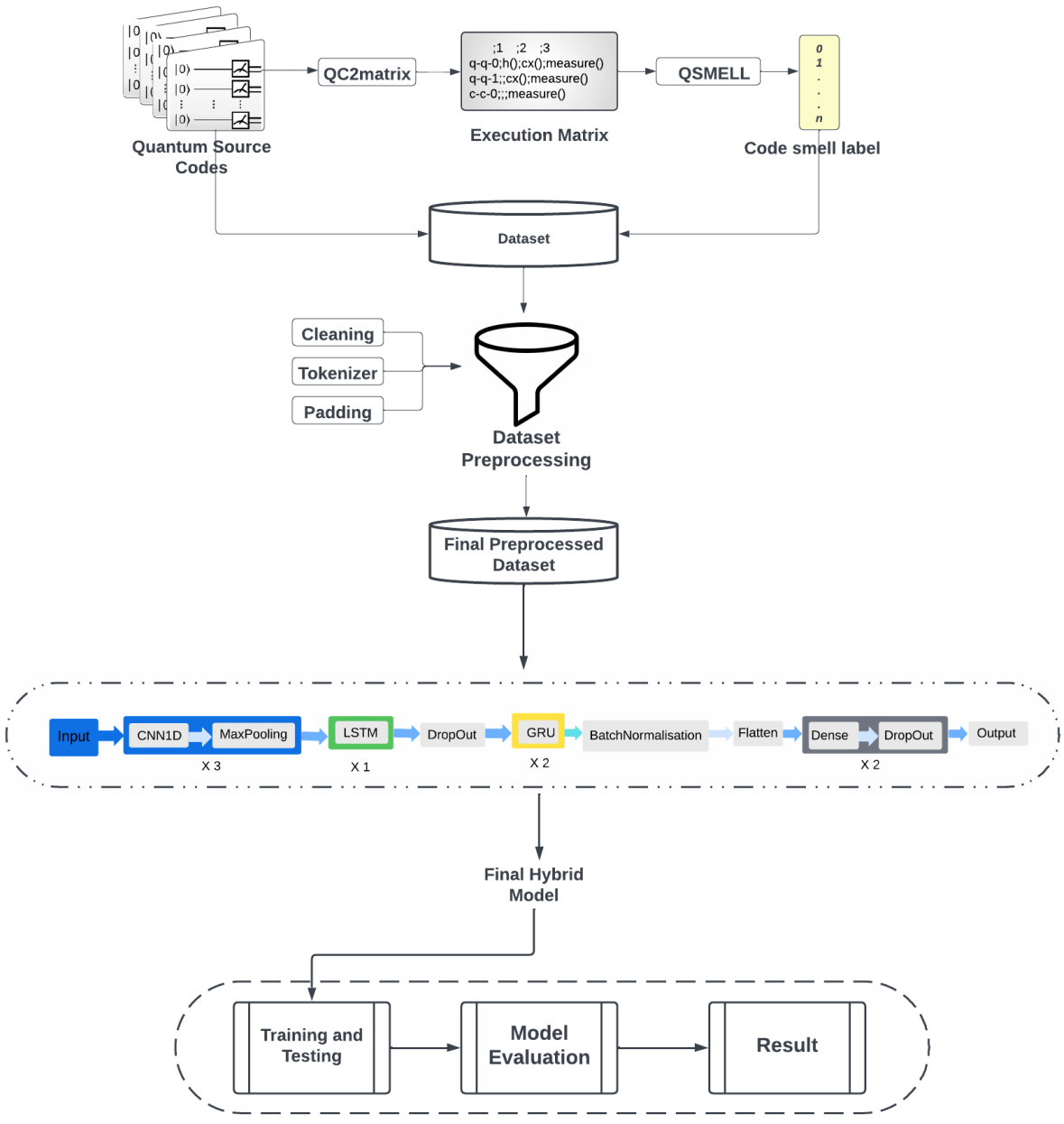


Figure 7.1: Overview of the Proposed Approach

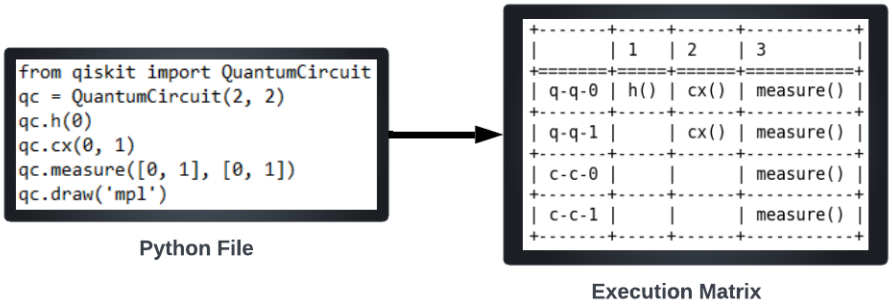


Figure 7.2: Converting Quantum Code to Execution Matrix

subsequent processes. The summary of the proposed methodology used in this study is shown in Figure 7.1. Quantum source codes are manually selected and then processed by the qc2matrix algorithm to generate an execution matrix. This matrix serves as the input data for QSmell used in this study, from which code smell labels are extracted. The combination of the source code and these labels constitutes the dataset, which is further pre-processed and provided as input data to the DL models.

7.2.2.1 Converting Quantum code to Execution Matrix

QPs are relatively different from regular programs coded in languages like Java and C++. QPs use quantum gates to build quantum circuits. These circuits process data and give the results through qubit phases. Detecting problems in quantum code requires understanding how these gates are used in the source code. QSmell, which works on the dynamic approach of QCS detection, is used to help find the QCS. For the input to the tool, an execution matrix is required to visualise the processes undergoing a quantum qubit. An example of conversion of a python file to an execution matrix using qc2matrix is shown in Figure 7.2. In this figure, each row represents a quantum bit (qubit) or a classical bit (bit), and each column represents a step in the process. Each cell in the chart shows a quantum operation that happened at a specific time, giving a detailed picture of how the circuit works. All the selected repositories were first converted into an execution matrix, and their output was saved in processable format to be passed to the QSmell tool.

7.2.2.2 Creating Labeled dataset

Creating a labeled dataset began with acquiring QSmell from its primary GitHub repository. Subsequently, a step-by-step analysis of each quantum code's execution matrix was conducted, utilizing QSmell to compute the values of these five parameters. This metic-

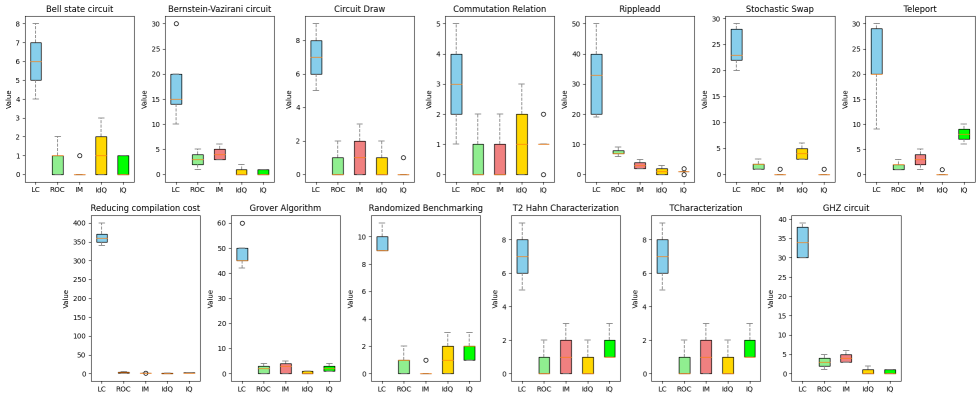


Figure 7.3: Values of QCS metrics for sample projects

ulous analysis generated parameter values and facilitated code classification and the creation of the manual dataset, which would serve as input data for the DL models applied in the study. For this study, distinct threshold values for each of the five parameters were established. These thresholds served as decisive benchmarks, guiding code classification into either smelly or non-smelly code. The threshold values taken for these parameters are mentioned in Equations 7.1 to 7.5 below:

$$LC(longCircuit) > 10 \tag{7.1}$$

$$ROC(RepeatedSetOfOperationsOnCircuit) > 0 \tag{7.2}$$

$$IM(IntermediateMeasurements) > 0 \tag{7.3}$$

$$IdQ(IdleQubits) > 0 \tag{7.4}$$

$$IQ(InitializationOfQubitsDifferentlyFrom|0\rangle) > 0 \tag{7.5}$$

If a quantum code satisfies one or more of the outlined criteria, it is classified as having a QCS. Conversely, a quantum code is considered non-smelly if it meets none of the provided criteria. To show the outcomes after QSmell detection, 13 sample projects are selected, and their outcomes are demonstrated in Figure 7.3. This figure displays various values observed ranging across specific parameters within each QCS.

7.2.3 Dataset Pre-processing

The dataset, formed in the preceding phase, undergoes pre-processing before being fed into the model for training. So, data pre-processing represents a pivotal stage in the overall process. The un-processed dataset has two columns: one with the raw Python scripts and the other with the binary rating of each code as smelly or not. Below are the steps

needed to pre-process this dataset.

7.2.3.1 Cleaning Code

Pre-processing this dataset is essential, and the initial step involves code cleaning. This process begins by eliminating all comments embedded within the QPs as shown in Eq. 7.6 and Eq. 7.7. By doing so, the model exclusively assesses the unadulterated functioning code, enabling a focused evaluation.

$$\text{cleanedCode} = \text{remove}(\text{start}(`\#'), \text{end}(`endline')) \tag{7.6}$$

$$\text{cleanedCode} = \text{remove}(\text{start}(```'), \text{end}(```')) \tag{7.7}$$

After removing the comments, a fundamental cleansing process ensued, including eliminating stop words and punctuation. This pruning ensured that only pertinent code segments remained for further analysis.

7.2.3.2 Tokenizer

Following the data cleansing process, the source code underwent tokenization, a pivotal step involving converting Python program text into numerical data, which is essential for model training. This approach is consistent with established practices in tokenization, ensuring that the same tokens consistently represent analogous words. This streamlined approach enhances the effectiveness of feature extraction from the data, aligning with the methodologies employed in prior research on tokenization techniques. This process ensures forming an accurate matrix that acts as input to the neural networks for classification purposes.

7.2.3.3 Padding

Given the diversity in code lengths across the Python files, the necessity arose for sequence padding to standardize dataset dimensions. This padding process was carried out seamlessly by padding to *max_length* following tokenization.

7.2.4 Analysis of DL Models

In the previous sections of this study, the dataset has been carefully prepared and pre-processed for input into a DL model. The final dataset visual is shown in Figure 7.4. This figure concisely represents the distribution of source code sizes and their corresponding frequencies. Now, the focus is on creating an efficient DL model for accurate predictions.

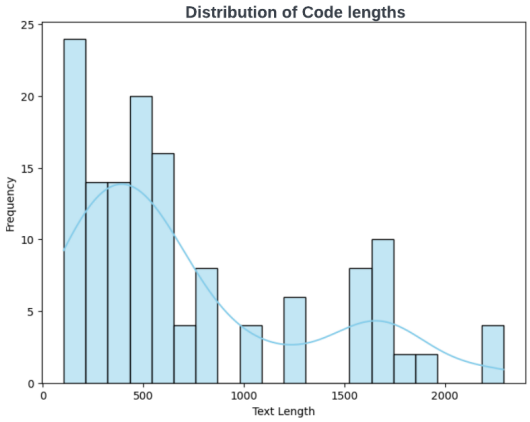


Figure 7.4: Analysis of Source Code Lengths

To achieve this, an ablation study is conducted to find the best DL models and their optimal settings. This study considers three neural network architectures: 1-D CNN, LSTM, and GRU. CNN is taken to be 1-D CNN, which suits the structure of the code dataset. These models are chosen for their unique abilities in feature extraction and prediction. Additionally, these models exhibit coordinated features when combined. The goal is to combine these models to create a hybrid model using the best parameters from each analysis.

7.2.4.1 Single Unit Ablation Study

Individual CNN, LSTM, and GRU models are constructed during the ablation study. It started with a baseline configuration, systematically increased the number of layers, and changed hyper-parameters while carefully monitoring performance. If the model produces satisfactory results initially, layers are incrementally added until a decrease in performance is observed. Hyperparameters were fine-tuned at each step, exploring combinations to find the best settings.

7.2.4.2 Hybrid Layer Ablation Study

This thorough ablation study helped identify the most effective CNN, LSTM, and GRU models. These models demonstrated their effectiveness individually on the curated dataset, paving the way for the next step: constructing a hybrid model. In this phase, two approaches were explored. First, the models were connected sequentially, one after the other, creating the "sequential hybrid model." Second, all individual models were run in parallel on the same dataset, and their outputs were merged to generate a single prediction, referred to as a "parallel hybrid model." Both hybrid models were evaluated inde-

pendently on the curated dataset, and a detailed comparison of their results is presented in the Results section. After conducting the analysis, the final "Proposed model" is formulated.

7.3 Results Specific to RQs

This section provides concise responses to all the research inquiries posed in this study, focusing on illuminating the significant outcomes and findings highlighted as the major contributions of this study.

7.3.1 RQ1: How do QCS influence the frequency of errors in Quantum computers?

In QC, noise emerges as a disruptive factor that compromises the stability of qubits, the fundamental units of quantum information. These disturbances lead to decoherence, wherein the qubit loses its delicate information over time. A variety of noise sources causes this effect; for example, IdQ introduces oscillations that cause qubit decoherence, compromising the integrity of encoded data. Even IQ contributes to decoherence by making qubits vulnerable to instability. Furthermore, the complexity of quantum algorithms with several gates can exacerbate mistakes via LC, resulting in deviations from the required quantum states. IM increases the measurement error, making the qubit unstable and collapse, and ROC can induce readout errors throughout the measurement process. These errors accumulate, leading to decoherence and changing the final results as shown in Figure 7.5. This diagram shows that a qubit should remain consistent without any changes in an ideal scenario.

This is ideal as qubits are unstable and possess more than one or partial state at a time. When gates are worked upon the qubits, it tends to change the qubit's state. When gates are not applied to these qubits, they still change their state, which can hinder the calculation done in that QP. This adds to the errors in the calculation, which are impacted or increased by the induction of QCS in these qubits. When operating within a quantum environment, various errors and QCS can lead to qubit decoherence, causing it to revert to its stable state $|0\rangle$ gradually. The figure illustrates how different types of QCS impact the qubit's behaviour. As depicted in the diagram, the presence of various types of QCS affects the stability of qubits. It's important to note that each QCS has a distinct impact. For instance, IM leads to a gradual change in qubit behaviour with fewer frequent errors, whereas LC and IQ introduce more pronounced variations characterised by frequent changes in qubit states.

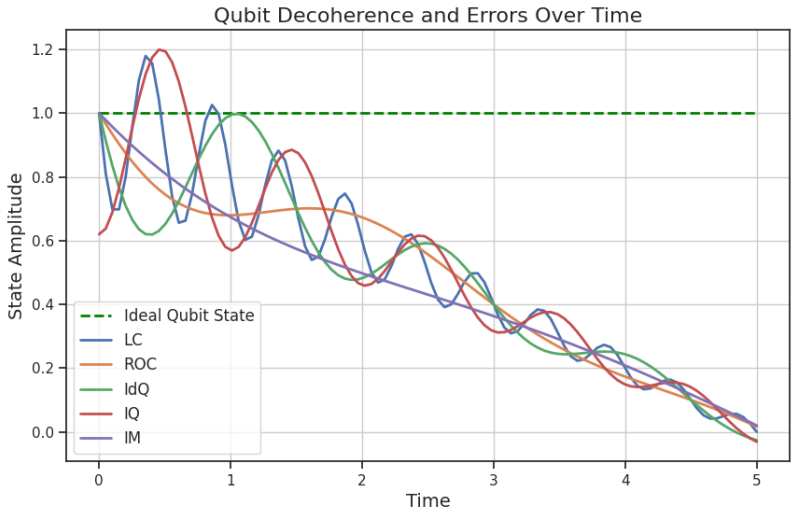


Figure 7.5: Error Produced due to Decoherence of Qubits

7.3.2 RQ2: What is the relative impact of different QCS on the performance of QC systems?

In QC, researchers are discovering how QCS in the code can make computer programs run slower and less reliably. Like classical computers, some code smells are more problematic than others, and their impact on the qubits varies with the threshold values. This is akin to how experts study computer problems in classical computers to improve programs.

In QC, it's crucial to identify which QCS have a more significant impact and which have a minor impact. This is similar to understanding which issues in classical computers are more serious. By doing this, a focus can be placed on fixing the significant issues first, making QP work better, and making them easier to manage. It is also important to note that various quantum computer types have unique characteristics. So, what may be a significant issue in one kind of quantum computer may be less effective in another.

By better understanding these issues, researchers can develop improved methods for detecting and correcting errors in quantum code. This will help quantum computers perform their tasks effectively and bring the research closer to using them for many exciting applications.

This research examined 136 projects and thoroughly recorded the occurrences of QCS within them. To provide insight into this section, a subset of 50 sample projects has been selected from the larger pool of 136 projects, and their data is presented for better comprehension. Table B.1 describes the values of these 50 sample projects. The table provides the established threshold values for each QCS.

Table 7.1: Evaluation of Individual models

Metric	CNN 1-D			LSTM			GRU		
	One Layer	Two Layers	Three Layers	One Layer	Two-Layers	Two-Layer (Bi-LSTM)	One Layer	Two Layers	Three Layers
Accuracy	82.1	78.0	85.7	91.9	86.0	61.0	89.0	89.3	86.0
Precision	68.0	69.0	76.9	93.0	90.0	37.0	92.0	92.0	90.0
Recall	100.0	81.0	90.9	93.0	86.0	61.0	89.0	89.0	86.0
F1-Score	81.5	75.0	83.0	93.0	87.0	46.0	92.0	92.0	90.0
MCC	69.7	71.7	73.8	85.0	74.0	19.0	80.4	80.4	74.9

Among these QCS, the most critical one in QP is IdQ. It stands out as the most severe because it triggers decoherence, introducing a substantial number of errors. Conversely, IM is considered the least impactful QCS due to its comparatively minor influence on the quantum system's stability and accuracy.

7.3.3 RQ3: Do the outcomes of DL models, developed with distinct hyper-parameters, exhibit complementary behavior?

In this RQ, various combinations of DL models have been evaluated by adjusting their architecture and hyperparameters. Specifically, three distinct types of individual models, CNN, LSTM, and GRU, have been considered. Each model has been assessed to determine the optimal number of layers that yield the best performance on the dataset.

7.3.3.1 Single Layer Ablation Study

Initially, each of the chosen models, namely CNN, LSTM, and GRU, has been evaluated independently to assess its efficacy in identifying QCS. The outcomes of the different CNN, LSTM, and GRU architectures are compared and illustrated in Table 7.1. It is evident from the results that the 1D CNN model with three layers consistently outperforms the other configurations. This observation can be attributed to the doubled number of filters applied in each layer of this CNN architecture, resulting in highly efficient feature extraction. Consequently, among the CNN models evaluated, the 3-layer 1D CNN emerges as the most promising choice for achieving accurate results.

Additionally, the performance of various GRU configurations is considered. It is evident from the findings that the GRU model with a single layer demonstrated promising results. Consequently, an experiment was conducted by increasing the number of layers to two, each containing 128 units. The results indicated that as the layers increased to

two, the performance remained comparable to that of the single-layer GRU, with only a marginal improvement.

However, upon further exploration with three layers, a notable decline in all evaluation metrics was observed. This resulted in the conclusion of the experiment after this stage. Therefore, the GRU model with two layers and 128 units in each layer emerged as the most suitable configuration among those tested.

Lastly, the outcomes for various LSTM configurations are measured. In the case of LSTM, the initial experiment with a single layer and 128 hidden units yielded impressive results. However, as the number of layers was increased, the performance began to deteriorate. An experiment combining LSTM with Bidirectional LSTM (Bi-LSTM) layers was also conducted to explore alternative architectures. Regrettably, the results obtained from the Bi-LSTM model did not meet the expectations. Consequently, based on this evaluation, it was concluded that the LSTM model with a single layer of 128 hidden units represents the most effective choice among the tested configurations.

7.3.3.2 Hybrid Layer Ablation Study

From the single-layer ablation study, the best layers in different DL algorithms were evaluated. This section will further test the best layers from the above results based on their combination working to detect QCS. For this evaluation, each configuration from CNN, LSTM, and GRU was coupled with one another to form different hybrid models. Keeping their feature extraction strength and model working enhancement with each other, three different combinations were selected for this evaluation. CNN+GRU, CNN+LSTM, and LSTM+Bi-LSTM were tested for the detection of QCS.

Table 7.2 describes the results of the following evaluation. The CNN+LSTM combination achieved the highest accuracy at 89.84%, outperforming the other models. Additionally, it exhibited superior precision (90.12%), an F1-Score of 85%, and an MCC value of 88%, making it a stable and compatible choice. The recall value, while slightly lower at 90%, was competitive and only marginally less than the highest value achieved by CNN+GRU at 91%.

Table 7.2: Evaluation of Combination of Models

Evaluation Metric	**CNN+GRU**	**CNN+LSTM**	**LSTM+Bi-LSTM**
Accuracy	86.87	89.84	87.09
Precision	88.0	90.12	89.0
Recall	91.0	90.0	86.0
F1-Score	80.0	85.0	82.0
MCC	79.0	88.0	75.6

These evaluations highlight the superiority of the CNN+LSTM hybrid model combination. These results can be attributed to the strengths of CNNs in capturing local patterns and spatial features and their ability to extract hierarchical features through convolutional and pooling layers.

LSTM models excel at modeling sequential dependencies in data, which is particularly important for time series and sequential data. Combining these two types of information proved advantageous, allowing the model to consider spatial relationships and temporal dependencies simultaneously in QCS detection.

Considering the features mentioned above and the results from training the QCS dataset with different model combinations, it becomes clear that CNN+LSTM is the optimal choice for creating an improved hybrid model for QCS detection. The findings from this section will be extended to further analysis by incorporating additional compatible layers, enhancing the model's complexity in pursuit of higher accuracy.

7.3.3.3 Proposed Model:

Based on the results from the previous section, it was evident that the combination of CNN and LSTM (CNN+LSTM) significantly improved QCS detection, achieving an accuracy of 89.84%. To build on this success, the next step involved sequentially adding a GRU layer to the CNN+LSTM hybrid model. This increased the model's complexity and enhanced its ability to capture long-range dependencies present in the dataset.

By combining CNN, LSTM, and GRU, the model exhibited an ensemble effect, with each component contributing its unique strengths and compensating for the weaknesses of the others. The goal was to explore different layer combinations to ultimately propose a model yielding the highest accuracy for QCS detection.

The detailed architecture of each layer in this proposed model is described in Table 7.3. It commences with a CNN as the initial network, featuring three layers for effective feature extraction. This is followed by an LSTM layer for modeling and integrating the extracted features. Finally, a GRU layer is added to enhance the dependency factor. The model includes max-pooling, dense, flattened, and dropout layers to ensure input data consistency at each layer. This model achieved an impressive accuracy of 92.86%, precision of 100%, recall of 81.81%, F1-Score of 90%, and MCC value of 85.56%, which is the highest among all the tested combinations.

Table 7.3: Architectural Details of the Proposed Model

Model	CNN 1D	GRU	LSTM
Layers	3	2	1
Architecture	Filters: 64, 128, 256 Kernel Size: 3	Units: 128	Hidden Units: 128
Learning Rate	0.001	0.001	0.0001
Dropout	0.5	0.2	0.5
Accuracy	85.71	89.28	91.86
F1-Score	83	86	93
Recall	90.9	89	93
Precision	76.9	92	93
MCC	73.76	80.43	85

7.3.4 RQ4: What are the predictive capabilities of different DL models developed for QCS detection?

From the ablation study, the parameters and layers showing the best accuracies were selected for each DL model, including CNN, LSTM, and GRU. These models were then used to construct the "Proposed Model." The final configuration of the proposed model is shown in Table 7.3. Remarkably, these models demonstrated strong performance during their ablation studies. In this RQ, all the models are compared to determine the best-performing model in predictive capabilities.

To achieve this, all of these models underwent testing on the test split dataset. Their performance was assessed, and their respective ROCC was generated, a critical tool for this Comparison. ROCCs provide valuable insights into the models' discriminative power across various classification thresholds. The Comparison based on ROCs is shown in Figure 7.6.

By examining these ROC curves, how well each model can distinguish between positive and negative samples can be visualized, revealing their varying degrees of discriminative ability. Additionally, AUC is a numerical metric for overall model performance, with higher AUC values signifying better discriminatory power. It is observed from the figure that the proposed model shows the best AUC value as **0.93**.

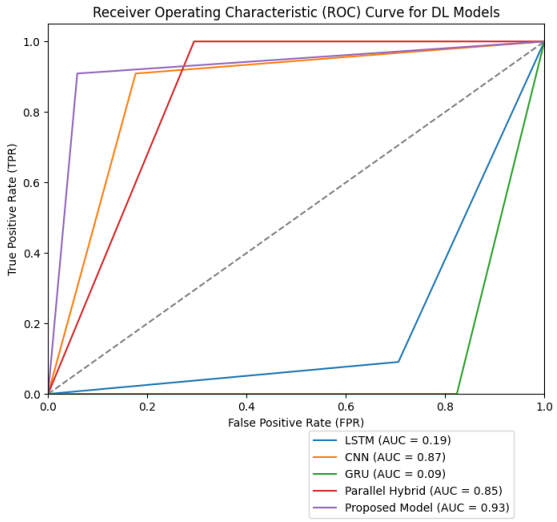


Figure 7.6: ROC curve comparison among DL models

7.3.5 RQ5: What is the effectiveness of the proposed model when compared with well-known DL models and pre-trained transfer learning model?

This RQ focuses on validating the novelty and credibility of the proposed model compared to other comparable and pre-trained models. To accomplish this, a comparative analysis of each model's performance was undertaken, leveraging diverse performance metrics. Each of these metrics plays a crucial role in the evaluation process, providing a comprehensive assessment of the models under examination.

This analysis compared independent CNN, LSTM, and GRU models to various hybrid models. The hybrid models included a parallel hybrid model and a pre-trained Bert model. These model results were compared with the proposed model introduced in this study.

The results are shown in Figure 7.7. All the models were evaluated using five metrics - accuracy, precision, recall, F1-score, and MCC. The values from each model were assessed for an overall comparison. Observations indicated that neither of the single-model structures outperformed the proposed model. When comparing the proposed model to the single-layer models, it became apparent that the complexity and enhancements in the proposed model significantly surpassed those of the individual models. Similarly, a hybrid parallel model was created based on combining individual layers. In this setup, each layer was granted the autonomy to extract features from the data independently, and the results were subsequently integrated and trained on the dataset. Notably, unlike the proposed model combination, where the models are connected one after the other,

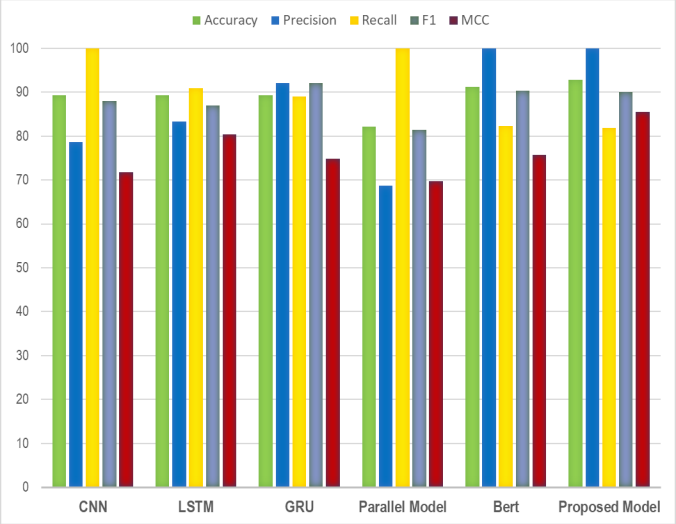


Figure 7.7: A Comparative Analysis of Results Between the Proposed Model and Other Models

the parallel architecture exhibited a distinct behaviour. In this case, the overall accuracy demonstrated a noticeable decrease, aligning with the findings observed across other result metrics.

To delve further into the details, it's worth noting that while the proposed model emphasizes a sequential flow of data, the parallel model enables each layer to work independently and parallelly. This shift in approach may have led to differences in feature extraction and how the models collectively interpret the data, ultimately impacting their overall performance.

These findings emphasize the significance of the architectural design choice, and they underline that the sequential and parallel model configurations can have varying impacts on the model's ability to understand and generalize from the dataset. The choice between these approaches should consider the data's specific characteristics and the analysis's goals.

A comparative analysis using transfer learning was conducted to enhance the diversity of testing for the proposed model. Specifically, this study utilized a specialized pre-trained BERT model that had already undergone training for classical computer code smell detection. This pre-trained BERT model was fine-tuned and evaluated on a quantum computer dataset. The results obtained from the BERT model were then compared to the proposed model. As depicted in this figure, it is evident that the BERT model outperformed the parallel model. Still, interestingly, the proposed model outshined all of these models, achieving an impressive accuracy rate of 92.86%. This suggests that the combination of feature extraction and specific input feature analysis within the proposed

model contributed to improved detection of QCS.

7.4 Discussion

In alignment with conventional software development practices, QC faces challenges related to the emergence of code smells, posing threats to long-term software maintainability. This research leverages advanced DL techniques to identify these QCS. Employing QSmell, a dedicated tool for quantum software analysis, and assembling a manual dataset comprising 136 QPs, this study establishes the potential of DL in the context of QC.

Moreover, this research introduces a thorough ablation study to craft the most precise DL model for QCS detection. Specifically, a sequential hybrid model is proposed, incorporating an individual CNN 1D model with three layers, an LSTM model with one layer, and a GRU model with two layers.

The Proposed model exhibited an impressive accuracy of 92.86%, underscoring its efficacy as an innovative approach. Additionally, the research uncovered intricate relationships between specific QCS and the errors they induce, providing insights into how these errors can compromise the maintainability of quantum code.

However, this study faced limitations regarding a small set of QCS and datasets due to the restricted availability of suitable QPs for a valuable dataset. Future endeavors aim to enhance QCS detection accuracy by utilizing an expanded dataset. The exploration of ***Generative AI*** for generating QPs datasets presents a promising avenue for future research. Furthermore, there is potential to extend the application of QSmell to QCS classification, transforming the model from a binary classifier into a multi-label classifier.

By demonstrating the feasibility of DL for QCS detection in the QC domain, this study not only paves the way for further exploration but also opens avenues for refining software quality assessment in the evolving field of QC.

Chapter 8

m-QIEA: An Optimized Approach for Code Smell Detection using Quantum-Inspired Evolutionary Algorithm

This chapter investigates a feature selection–driven approach for code smell detection in both classical and quantum software systems. The proposed approach leverages DL models for feature extraction, followed by a modified Quantum-Inspired Evolutionary Algorithm (m-QIEA) to refine the feature subset for classification. Compared with standard QIEA, m-QIEA incorporates a time-decaying rotation schedule and Gaussian mutation, which are intended to mitigate premature stagnation during the feature-selection search. Experimental evaluations are conducted on a comprehensive suite of six datasets. For classical systems, the primary MLCQ benchmark is used and further the work is extended to SmellyCode++, and a multi-language corpus (Python, Java, and C++). For quantum systems, the primary QNet benchmark is used, extended to a derived large-scale LintQ corpus, and tested on the independent Smelly Eight benchmark. Across all settings, the results demonstrate that m-QIEA achieves superior accuracy and generalization. The study establishes the critical role of FS in optimizing code smell detection accuracy. By integrating DL for feature extraction and m-QIEA for FS, the proposed approach enhances the classification of code smells across multiple programming paradigms and dataset scales. The remainder of this chapter is structured as follows: Section 8.1 gives the introduction of this chapter. Section 8.2 outlines the proposed methodology, presenting a detailed account of the steps undertaken in this study. Section 8.3 discusses the results. Finally, Section 8.4 discusses the findings of the chapter and highlights the guidelines for future work.

8.1 Introduction

In its early stages, code smell detection relied on manual efforts, evolving into a heuristic-based approach with pre-designed detection rules for specific code smells [247, 248, 33]. However, the heuristic-based approach proved inconsistent in many cases [249, 250]. This limitation prompted the development of ML approaches for detection [34]. The primary advancement in this field involved the creation of various models with feature development as a central focus [145, 203, 168, 141]. In the realm of ML, manual feature passing was a cumbersome task. Consequently, researchers transitioned from traditional ML methods [251] to the integration of DL techniques [166]. These DL models effectively handled extensive datasets by autonomously developing features across different layers [141, 46]. Nevertheless, the adoption of DL may give rise to irrelevant features, posing a potential compromise to the model's training capacity. In response to this challenge, FS has become a critical research area, with the objective of optimizing or sub-optimizing feature subsets based on evaluation criteria [252]. FS plays a crucial role by not only enhancing classifier efficiency through the elimination of unimportant features but also improving clarity and addressing overfitting concerns through the removal of noisy data [253].

Recent studies have delved into the integration of various evolutionary optimizers in FS, showcasing the reliability of metaheuristic algorithms in addressing complex optimization problems like data clustering and FS [159]. While traditional evolutionary algorithms such as GA [254, 255] and PSO [256] are widely applied in optimization tasks, their flexibility is often hindered by prolonged processing times, limiting their practicality in classification contexts. Additionally, these algorithms may encounter challenges related to getting stuck in local optima, impeding their convergence to the optimal solution.

To overcome these limitations, a groundbreaking approach called the Quantum Inspired Evolutionary Algorithm (QIEA) was introduced [2]. QIEA stands out for its efficiency and reduced computational time. Numerous studies have explored the application of QIEA, incorporating specific modifications for tasks like fuzzy classification and image prediction. However, to the best of the authors' knowledge, there has been no reported utilization of QIEA in the domain of code smell detection. Hence, the primary objective of this study is to introduce a novel modified quantum-inspired evolutionary-based FS method **(m-QIEA)** designed specifically for code smell detection. The proposed methodology begins by extracting features from the second last layer of three distinct DL models. Subsequently, m-QIEA is applied to derive the optimal feature subset from the extracted features. Finally, the identified feature set is fed into ML classifiers for code smell classification.

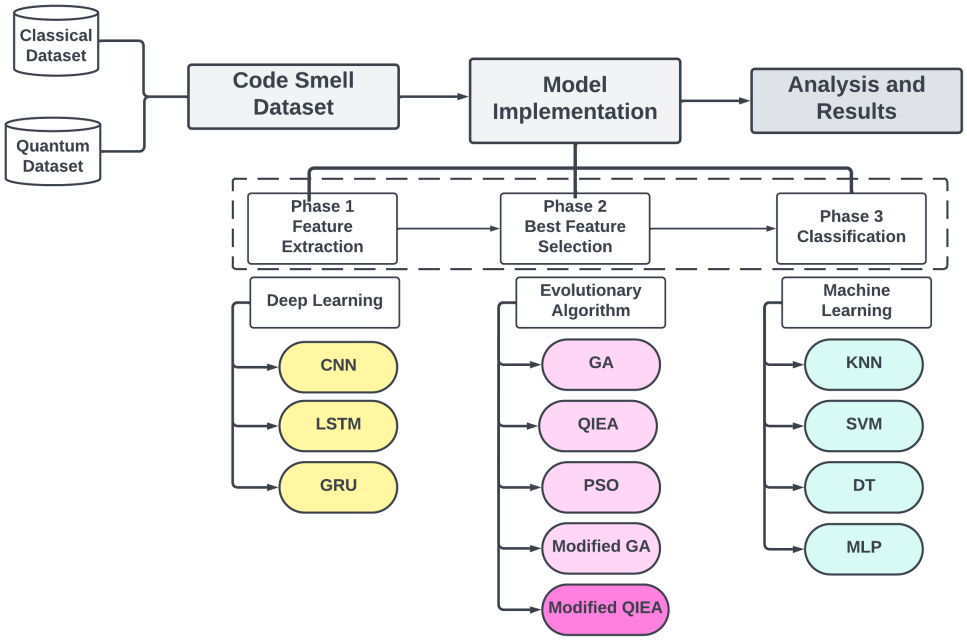


Figure 8.1: Overview of Proposed Methodology for Code Smell Detection

This proposed approach is tested on a structured suite of six datasets to ensure robustness across scales, languages, and quantum platforms. This multi-tiered validation strategy rigorously tests the approach's generalizability, ensuring that the results are not confined to a single language or dataset size. The significance of this paper lies in its substantial contribution to the field by automating the feature extraction process for code smell detection in both quantum and classical systems. This marks a pioneering effort in advancing the methodologies employed in this domain, backed by extensive empirical validation across diverse software ecosystems.

8.2 Experimental Framework

This research delves into code smell detection through FS domain exploration. The flow diagram of the proposed approach is depicted in Figure 8.1. The methodology is validated on two categories of datasets: classical and quantum, undergoing model implementation in three phases. First, LSTM, CNN, and GRU DL models extract features. Features are extracted from the second-to-last layer to ensure consistency, aiding subsequent FS. In the second phase, extracted features undergo optimization using GA, QIEA, PSO, Modified GA, and m-QIEA algorithms as described in section 2.5.3. These converge cost functions to produce specific optimal feature sets. Optimal features are then inputted into KNN, SVM, DT, and MLP classifiers for validation in the final phase.

8.2.1 Dataset

This section provides a comprehensive overview of the datasets used in this study, detailing their composition, selection criteria, and relevance to the research. The datasets include both classical and quantum program repositories, carefully curated to facilitate code smell detection and analysis. The experimental validation incorporates six datasets organized across classical and quantum domains: two primary benchmarks (MLCQ and QNet) and four supplementary datasets (SmellyCode++, Multi-language Code Smells Instances, derived LintQ corpus, and Smelly Eight benchmark). This multi-dataset strategy ensures robust evaluation across varying scales, programming languages, and application domains.

Table 8.1 provides a structured overview of all six datasets, summarizing their key characteristics including language, size, smell types, labeling approach, and intended validation purpose.

Table 8.1: Summary of Datasets Used in the Study

Dataset	**Language**	**Samples**	**Code Smells Covered**	**Labeling Approach**	**Validation Purpose**
MLCQ	Java	2500	Feature Envy, Blob, God Class, Large Class	Manual review by 26 expert developers	Primary classical benchmark
SmellyCode++	Java	107,554	God Class, Data Class, Feature Envy, Long Method	Multi-label annotation from AST extraction	Large-scale classical validation
Multi-language Corpus	Java, Python, C++	3,470	Consistent smell definitions across languages	Expert-labeled cross-lingual instances	Cross-language generalization
QNet	Python (Qiskit)	136	Six quantum-specific smells (LC, IM, IQ, IdQ, ROC, CG)	QSMELL tool with manual curation	Primary quantum benchmark
Derived LintQ Corpus	Python (Qiskit)	7,118	Six quantum-specific smells (same as QNet)	QSMELL tool with inter-rater validation ($\kappa = 0.89$)	Large-scale quantum validation
Smelly Eight	Python (Qiskit)	15	Eight quantum-specific smells	Manual curation by quantum experts	Independent external quantum test

8.2.1.1 Classical Datasets

The proposed method is initially evaluated on the classical *MLCQ* benchmark. The dataset comprises nearly 2,500 labeled Java code samples from 792 real-world projects, covering four code smell types: *Feature Envy, Blob, God Class*, and *Large Class* [257]. Each record in the MLCQ dataset contains the type of smell, its severity, and the exact location in the source code, enabling detailed analysis of code smell characteristics. Expert developers reviewed the samples to ensure label quality and severity assessment. Its widespread adoption in prior research underscores its significance as a benchmark for code smell detection and deep learning applications [258].

Furthermore, two supplementary classical datasets are incorporated to evaluate scalability and cross-language generalizability. *SmellyCode++* [259] is a large-scale Java dataset with 107,554 samples and multi-label annotations for God Class, Data Class, Feature Envy, and Long Method. The samples were extracted by parsing code into Abstract Syntax Trees (ASTs), extracting features, and annotating based on prior studies. *Multi-language Code Smells Instances* [260] contains 3,470 samples across Java (1,240), Python (1,180), and C++ (1,050) with consistent smell definitions across languages.

8.2.1.2 Quantum Datasets

The proposed method is evaluated on the quantum *QNet* dataset, comprising 136 Python-based quantum programs from real-world projects across seven categories. Programs are labeled using the *QSMELL* tool, which detects six quantum-specific smells [246, 261]. The resulting labeled dataset provides representative instances of smelly and non-smelly code, enabling a reliable and reproducible evaluation of the proposed approach in the quantum software domain.

Two supplementary quantum datasets were also used to broaden the validation scope. *LintQ Corpus* [132] is a large-scale collection of 7,568 real-world Qiskit-based quantum programs assembled for static analysis of quantum code quality. The programs were collected from open-source quantum software repositories and represent diverse application categories. For this study, the corpus was processed by removing exact and near-duplicates, discarding unparsable or version-incompatible files, and applying the same QSMELL configuration used for QNet. This process yielded 7,118 valid quantum programs with binary smell labels. A stratified sample of 400 programs was manually inspected by two independent reviewers, achieving an inter-rater agreement of 0.89 (Cohen's Kappa), ensuring label quality. This dataset enables evaluation of the proposed approach's scalability to large quantum codebases.

Smelly Eight Benchmark [70] comprises 15 manually curated quantum programs from

sources independent of QNet. The original study defined eight quantum-specific code smells, validated them through a survey with 35 quantum developers, and developed the QSMELL tool for detection. The empirical investigation found that 11 out of 15 programs (73.33%) contain at least one smell, with an average of three smells per program. This independent benchmark serves as an external test set to validate the generalizability of the proposed approach beyond the training distribution.

By incorporating both classical and quantum datasets, this study broadens the applicability of the proposed approach, enabling a comprehensive analysis of code smells across different computing paradigms

8.2.2 Phase 1: Feature Extraction

The datasets undergo a novel approach using DL models for feature extraction, diverging from their typical use for classification. Instead of generating classification output, the focus is on extracting features. During training, the dataset serves as input, and features are extracted from the second-to-last layer of the model. These features are saved for later use in optimal FS.

This study employs three fundamental DL classifiers – LSTM, CNN, and GRU, with their selected architectural details given in Table 8.2. The choice of these classifiers is motivated by their versatility and widespread applicability in various domains.

In preliminary experiments, GRU was identified as the optimal feature extractor for both classical and quantum code smell datasets. Table 8.3 and Table 8.4 summarizes the comparison of CNN, LSTM, and GRU across both categories of datasets, respectively. GRU consistently outperforms CNN and LSTM across both datasets, achieving the highest accuracy of 85.3% on classical and 92.5% on quantum datasets. Consequently, all subsequent feature selection experiments use GRU-extracted features unless otherwise noted.

The feature extraction process entails configuring the feature count to 512 for all three models. This decision is based on setting the final layer of each model to extend up to 512 units. Attempts to increase the model beyond this threshold resulted in overfitting. Therefore, 512 is considered the maximum number of features selected in the last units of the models. During training, the models undergo 100 epochs, a value chosen based on preliminary trials [262] that showed consistent convergence and performance stability within this range. This choice is grounded in prior empirical studies that have successfully used similar limits. Fontana et al. [263] and Di Nucci et al. [264] demonstrated the effectiveness of bounded iteration counts in machine learning and software engineering prediction tasks, including code smell and bug detection. Furthermore, domain-specific studies such as those by Saranya et al. [265] and Draz et al. [266] also employed 100

Table 8.2: Configuration of the Selected Models

Model	Hyperparameters	Value
CNN	Filters (Conv1D)	64
	Kernel Size (Conv1D)	3
	Dense Layer Units	512
	Final Dense Layer Units	1
LSTM	No. of Units	128, 512
	Dropout rate	0.2
	Learning Rate	0.001
GRU	GRU Units	1, 2
K-Nearest Neighbors (KNN)	n_neighbors	5
Support Vector Machine (SVM)	degree	3
	cache_size	300
	tol	0.0001
Decision Tree (DT)	criterion	Gini
	max_depth	3
	min_samples_leaf	2
Multi-Layer Perceptron (MLP)	num_layers	10
	learning rate	0.001
	batch_size	32, 64, 128

generations in their use of PSO and Whale Optimization Algorithm for code smell identification. As stated by Imam et al. [161], setting a predetermined number of generations is a straightforward approach that balances computational efficiency with solution quality. In the context of m-QIEA, experimental observations confirmed that meaningful convergence typically occurred well before 100 generations. Therefore, setting the limit at 100 offers a practical trade-off between optimization quality and computational cost. A batch size of 128 and a validation split of 20% is employed to assess the model's performance on a validation set.

8.2.3 Phase 2: Best Feature Selection

The extracted features are provided to the evolutionary optimizers as candidate inputs for feature selection. Each candidate solution is represented by a binary feature mask. An SVM classifier is used as the wrapper evaluator, and the optimizers search for the feature mask that minimizes the SVM prediction error measured using **Mean Squared Error (MSE)**. MSE acts as the convergence metric, which quantifies the average squared difference between the actual and predicted classification outcomes. The number of selected features is reported as an additional property of the best-observed mask; feature-subset

Table 8.3: Comprehensive Feature Extraction Performance Across Classical Datasets

Dataset	DL Model	Code Smell	Accuracy	Precision	F1-Score
MLCQ	CNN	Blob	62.37	68.23	62.76
		Feature Envy	60.4	70.5	62.9
		God Class	66.26	69.55	65.21
		Large Class	61.78	70.5	60.4
	LSTM	Blob	66.26	69.55	65.21
		Feature Envy	66.45	69.42	63.1
		God Class	67.45	72.54	68.0
		Large Class	62.37	71.5	62.9
	GRU	Blob	72.54	71.25	69.0
		Feature Envy	67.45	73.4	68.0
		God Class	69.2	70.5	62.9
		Large Class	66.45	69.55	65.21
SmellyCode++	CNN	God Class	71.2	72.8	72.1
		Data Class	70.5	71.9	71.3
		Feature Envy	69.8	71.2	70.5
		Long Method	70.1	72.0	71.0
	LSTM	God Class	72.8	73.5	73.1
		Data Class	72.1	73.0	72.5
		Feature Envy	71.5	72.8	72.0
		Long Method	71.8	73.2	72.3
	GRU	God Class	**83.8**	**85.2**	**86.5**
		Data Class	**82.5**	**84.0**	**84.8**
		Feature Envy	**83.2**	**84.5**	**85.9**
		Long Method	**83.5**	**84.8**	**86.0**
Multi-language	CNN	God Class	70.8	72.1	71.5
		Feature Envy	71.2	72.5	71.8
		Large Class	70.5	71.8	71.0
		Long Method	71.0	72.2	71.5
	LSTM	God Class	72.5	73.8	73.0
		Feature Envy	72.8	74.0	73.5
		Large Class	72.0	73.5	72.8
		Long Method	72.5	73.8	73.2
	GRU	God Class	**84.1**	**85.5**	**86.0**
		Feature Envy	**85.3**	**86.5**	**87.0**
		Large Class	**83.5**	**84.8**	**85.5**
		Long Method	**84.0**	**85.2**	**86.0**

Table 8.4: Comprehensive Feature Extraction Performance Across Quantum Datasets

Dataset	DL Model	Quantum Code Smells (F1-Score / Accuracy)					
		LC	**IM**	**IQ**	**IdQ**	**ROC**	**CG**
QNet	CNN	81.2/83.5	79.4/81.8	82.1/84.2	78.5/80.9	80.3/82.7	77.8/80.1
	LSTM	84.5/86.2	82.8/84.9	85.2/87.1	81.9/84.0	83.6/85.8	80.5/82.9
	GRU	**88.7/90.1**	**86.5/88.2**	**89.2/91.0**	**85.8/87.6**	**87.4/89.3**	**84.2/86.5**
Derived LintQ	CNN	84.3/86.0	82.1/84.2	85.0/86.8	81.5/83.6	83.2/85.4	80.8/83.0
	LSTM	86.8/88.5	85.0/87.1	87.5/89.2	84.2/86.3	85.8/87.9	83.5/85.7
	GRU	**90.2/91.8**	**88.4/90.0**	**91.0/92.5**	**87.5/89.2**	**89.0/90.8**	**86.5/88.2**
Smelly Eight	CNN	78.5/80.0	76.2/78.5	79.0/80.8	75.5/77.8	77.2/79.5	75.0/77.2
	LSTM	81.0/82.8	79.5/81.5	81.8/83.5	78.5/80.5	80.0/82.2	77.5/79.8
	GRU	**84.5/86.0**	**82.8/84.5**	**85.2/87.0**	**82.0/84.0**	**83.5/85.2**	**81.0/83.0**

Note: LC: Long Circuit; IM: Intermediate Measurements; IQ: Initialization of Qubits Differently; IdQ: Idle Qubits; ROC: Repeated Operations on Circuit; CG: Customized Gates. Values are reported as F1-score/accuracy.

size is not included as a separate term in the optimization objective. The MSE is computed as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{8.1}$$

where:

- n is the total number of samples,
- y_i represents the actual classification labels,
- $\hat{y}_i$ denotes the predicted labels.

A patience-based objective-stagnation criterion is used. The optimization terminates when the absolute change in the best MSE value over k iterations falls below the tolerance δ:

$$|MSE_t - MSE_{t-k}| < \delta \tag{8.2}$$

In the context of code smell detection, particularly when handling high-dimensional feature spaces present in both classical and quantum datasets, the optimization landscape can exhibit local fluctuations in error values. A fixed-threshold approach often fails to distinguish between genuine convergence and temporary stagnation, potentially leading to either premature halting or unnecessary iterations. To address this, the values of the convergence parameters were carefully selected through empirical tuning and guided by recommendations in evolutionary optimization literature:

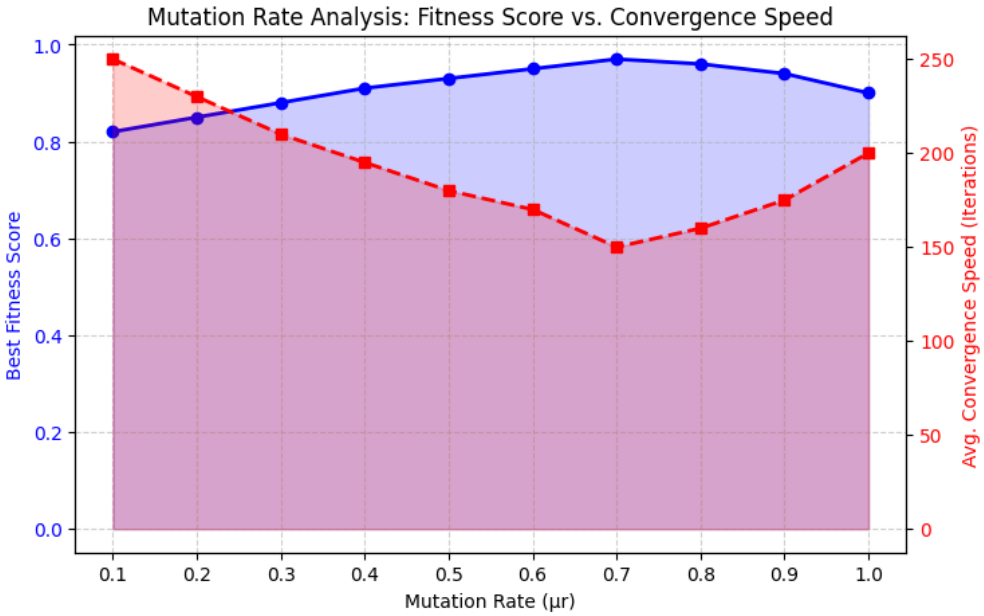


Figure 8.2: Different mutation rates on fitness score and convergence speed

- The threshold value δ is set to 1×10^{-4}, which offers a suitable balance between sensitivity and robustness. Given that MSE values typically lie within the range $[0, 1]$, this choice ensures that only negligible improvements in model performance trigger convergence, while minor variations due to noise or floating-point imprecision are ignored.

- The patience window size k is set to 5, enabling the algorithm to assess the stability of MSE over a meaningful horizon. A window of this size captures consistent behavior rather than reacting to single-iteration anomalies, which is crucial when dealing with datasets that exhibit high dimensionality and class imbalance.

This adaptive strategy contributes to a more reliable convergence behavior of the optimizer and facilitates the discovery of robust and selected feature subsets relevant for precise code smell detection across both classical and quantum domains.

Evolutionary optimizers crucially enhance the weights of features influencing model accuracy. Over 100 generations, the optimizers refine the feature subset, minimizing classifier error acting as the cost function. They also record the model's best accuracy score. This iterative process prioritizes features with optimized weights, aiming to select the most influential ones for an accurate model. Three standard evolutionary optimizers viz. GA, PSO, and QIEA are employed along with a modified version of GA [119]. After leveraging these optimizers to identify the best-selected features, a modified version of QIEA, referred to as *m-QIEA*, is introduced and elaborated upon in Section 8.2.3.1.

8.2.3.1 The Proposed m-QIEA

The proposed m-QIEA modifies the Q-bit update stage of standard QIEA while retaining its probabilistic binary representation. The modification integrates the rotational quantum gate with an iteration-dependent, time-decaying rotation scaling mechanism. The pseudo-code detailing the modified update process is provided in Algorithm 5. In this study, the binary value 1 denotes a selected feature, whereas 0 denotes an excluded feature. For notational consistency, $F(X)$ denotes a fitness function to be maximized. Since the wrapper evaluation is based on classification error, the maximized fitness is defined as $F(X) = -\text{MSE}(X)$, such that a larger value of F represents a better feature subset.

To preserve the Q-bit normalization constraint during both rotation and mutation, each Q-bit is parameterized by a real-valued angle. For the p-th feature of individual i at iteration t:

$$q_{i,p}^{(t)} = \begin{bmatrix} \alpha_{i,p}^{(t)} \\ \beta_{i,p}^{(t)} \end{bmatrix} = \begin{bmatrix} \cos\left(\theta_{i,p}^{(t)}\right) \\ \sin\left(\theta_{i,p}^{(t)}\right) \end{bmatrix}. \tag{8.3}$$

It follows directly that $\left|\alpha_{i,p}^{(t)}\right|^2 + \left|\beta_{i,p}^{(t)}\right|^2 = 1$. Therefore, rotation and mutation can be applied directly to $\theta_{i,p}$ without separately reconstructing or clipping the probability amplitudes.

The update process in m-QIEA involves determining the rotation angle for each quantum bit based on a condition affecting a randomly generated value and a mutation rate threshold (μ_r). A mutation rate of 0.7 was established through experimentation. Figure 8.2 presents the experimental results, illustrating the relationship between mutation rate, best fitness score, and convergence speed. The study observed that a mutation rate of 0.7 resulted in the highest fitness score while maintaining a favorable convergence speed. Beyond this threshold, fitness scores exhibited marginal improvements or slight degradations while convergence speed began to deteriorate. In Algorithm 5, μ_r directly denotes the probability of executing the mutation branch. Since rand_{pcg} is uniformly distributed over $[0, 1)$ and mutation is performed when $\text{rand}_{\text{pcg}} < \mu_r$, the probabilities of the two update branches are:

$$\Pr(\text{mutation}) = \mu_r, \qquad \Pr(\text{rotation}) = 1 - \mu_r.$$

Thus, $\mu_r = 0.7$ implies that mutation is applied to approximately 70% of Q-bit updates, while rotation is applied to approximately 30%.

Before updating the Q-bits, the best solution is updated when the currently observed solution is better:

$$F\left(X_i^{(t)}\right) > F\left(B^{(t)}\right) \quad \Longrightarrow \quad B^{(t)} \leftarrow X_i^{(t)}.$$

Algorithm 5 The Proposed m-QIEA Updation Process

For each Quantum bit (q_i) defined as $\{q_1|q_2|\ldots|q_p|\ldots|q_m\}$ where $q_p = [\alpha_p \beta_p]^T$
Initialize $\mathrm{rng} \leftarrow \mathrm{Generator}(\mathrm{PCG64}(\mathrm{seed}))$
while $i \leq m$ **do**
    if $\mathrm{Better}(x, b)$ **then**
        $g_i \leftarrow x_i$
    else
        $g_i \leftarrow b_i$
    end if
    if $x_i = b_i$ **then**
        $\Delta\theta_i \leftarrow 0$
    else if $g_i = 1$ **then**
        $\Delta\theta_i \leftarrow 0.005\pi \ \mathrm{sgn}_+(\alpha_i^{(t-1)} \beta_i^{(t-1)})$
    else
        $\Delta\theta_i \leftarrow -0.005\pi \ \mathrm{sgn}_+(\alpha_i^{(t-1)} \beta_i^{(t-1)})$
    end if
    $\mathrm{rand}_{\mathrm{pcg}} \leftarrow \mathrm{rng.random}()$
    if rand_pcg $\geq \mu_r$ **then**
        Perform Quantum Rotation Gate:
        $\widetilde{\alpha}_i^{(t)} = \cos(\lambda(t)\Delta\theta_i)\alpha_i^{(t-1)} - \sin(\lambda(t)\Delta\theta_i)\beta_i^{(t-1)}$
    else
        Perform Mutation: $\widetilde{\alpha}_i^{(t)} = \alpha_i^{(t-1)} + \mathcal{N}(0, \sigma^2)$
    end if
    $\alpha_i^{(t)} \leftarrow \min\{1, \max\{-1, \widetilde{\alpha}_i^{(t)}\}\}$
    $s_i \leftarrow \begin{cases} +1, & \beta_i^{(t-1)} \geq 0 \\ -1, & \beta_i^{(t-1)} < 0 \end{cases}$
    $\beta_i^{(t)} \leftarrow s_i \sqrt{\max\{0, 1 - (\alpha_i^{(t)})^2\}}$
    Update the Q-bit: $q_i^{(t)} \leftarrow [\alpha_i^{(t)}, \beta_i^{(t)}]^T$
end while

Table 8.5: Direction rule for the m-QIEA rotation update. The condition $F(X_i) < F(B)$ indicates that the stored best solution has better fitness than the current observed solution.

$x_{i,p}$	b_p	$F(X_i) < F(B)$	$d_{i,p}$
0	0	true	0
0	0	false	0
0	1	true	$+\operatorname{sgn}_0(\alpha_{i,p}\beta_{i,p})$
0	1	false	0
1	0	true	$-\operatorname{sgn}_0(\alpha_{i,p}\beta_{i,p})$
1	0	false	0
1	1	true	0
1	1	false	0

The signed rotation direction for each feature depends on the observed bit $x_{i,p}^{(t)}$, the corresponding best-solution bit $b_p^{(t)}$, the relative fitness of the two solutions, and the current Q-bit quadrant. For concise notation:

$$\operatorname{sgn}_0(z) = \begin{cases} +1, & z \geq 0, \\ -1, & z < 0. \end{cases} \tag{8.4}$$

The factor $\operatorname{sgn}_0(\alpha_{i,p}^{(t)}\beta_{i,p}^{(t)})$ identifies the rotation direction that increases the probability of the desired target state. Table 8.5 defines the complete direction rule used in m-QIEA.

When $x_{i,p} = b_p$, no directed rotation is required. When $x_{i,p} \neq b_p$ and the stored best solution has higher fitness, the rotation is directed toward b_p. For $b_p = 1$, the update increases $\Pr(x_{i,p} = 1) = |\beta_{i,p}|^2$; for $b_p = 0$, it increases $\Pr(x_{i,p} = 0) = |\alpha_{i,p}|^2$. If the current solution is better, it is stored as the new best solution, and no rotation toward the previous inferior solution is performed.

The magnitude of the directed rotation is controlled by a time-decaying schedule:

$$\lambda(t) = \max\left\{0,\, 1 - \frac{t}{T}\right\}, \qquad 0 \leq t \leq T, \tag{8.5}$$

where t is the current iteration and T is the maximum number of iterations. The effective rotation magnitude is $\lambda(t)\Delta\theta_0$, where $\Delta\theta_0$ is the base rotation magnitude. Larger rotation steps are used during the early search, and progressively smaller steps are used as the iteration count increases. This schedule permits comparatively larger exploratory steps during early iterations and refined exploitation during later iterations.

When the random value falls below μ_r, the quantum bit is mutated. Unlike standard QIEA, m-QIEA introduces a localized mutation using Gaussian noise:

$$\alpha_i^{(t)} = \alpha_i^{(t-1)} + \mathcal{N}(0, \sigma^2), \tag{8.6}$$

where σ is typically set to 0.1. This maintains the continuity of promising solutions by introducing minor perturbations rather than radical changes. In code smell detection, where software quality metrics often exhibit subtle nonlinear relationships, controlled variation via Gaussian noise supports better exploitation without sacrificing convergence stability.

Because Gaussian mutation can produce candidate amplitudes outside $[-1, 1]$, $\tilde{\alpha}_i^{(t)}$ is first clipped to this interval. The corresponding $\beta_i^{(t)}$ is reconstructed using:

$$\beta_i^{(t)} = s_i \sqrt{\max\left\{0, 1 - \left(\alpha_i^{(t)}\right)^2\right\}}, \tag{8.7}$$

where $s_i = +1$ when $\beta_i^{(t-1)} \geq 0$ and $s_i = -1$ otherwise. Retaining the sign of the previous β amplitude preserves the intended Q-bit quadrant. The $\max\{\cdot\}$ term prevents a negative square-root argument caused by floating-point rounding.

This integration of time-decaying rotation and controlled mutation enhances m-QIEA's ability to balance exploration and exploitation, mitigating the risk of convergence to local optima. The algorithm is further supported by the use of the PCG64 pseudo-random number generator in NumPy, initialized with a fixed seed for reproducibility.

In summary, m-QIEA offers the following benefits in FS for code smell detection:

- m-QIEA introduces an adaptive approach based on the current state of the Qubit. The positive/negative $\Delta\theta$ values in different quadrants allow for more directed exploration in promising areas while still enabling the exploration of new possibilities. The comparison with μ_r creates a balance between exploration and exploitation. This can be particularly beneficial in the context of code smell detection, where the optimal feature set might involve a combination of widely relevant and more specific features.

- m-QIEA uses a smaller, fixed value (0.005π) for $\Delta\theta$. This allows for more controlled rotations, potentially leading to a more gradual exploration of the search space. This is advantageous in code smell detection, where small changes in feature selection can significantly impact the outcome.

- m-QIEA scales the rotation angle through the time-decaying function $\lambda(t)$, enabling comparatively larger search steps during early iterations and smaller refinement steps during later iterations. This enhances convergence and avoids premature stagnation.

- m-QIEA introduces Gaussian-based mutation rather than fully random re-initialization, preserving the neighborhood of high-quality solutions and improving the local search

behavior near optima. This is especially relevant for code smell detection, where preserving semantically meaningful feature combinations is critical for classification accuracy.

- m-QIEA has the potential to converge on the optimal feature set faster than the standard QIEA. This can be crucial for real-world applications where processing large codebases is necessary.

8.2.4 Phase 3: Classification

The optimal features identified by the optimizers are employed to train four distinct ML classifiers: KNN, SVM, DT, and MLP. Each classifier introduced a distinctive methodology to the task, showcasing the versatility of the applied techniques. KNN operates by classifying data points according to the majority class among their nearest neighbours in the feature space. In contrast, SVM diligently sought optimal decision boundaries, strategically maximizing the margin between different classes. DT took a recursive approach, systematically splitting the dataset to construct DTs, effectively capturing intricate relationships within the data. MLP, a sophisticated artificial neural network, demonstrated its capability to learn intricate patterns through interconnected layers, adapting its internal weights during the training process. The results, derived from the diverse output of these classifiers, not only showcased their strengths but also facilitated a thorough evaluation of their collective performance within the feature extraction and classification pipeline.

Table 8.6: Performance Comparison of Machine Learning Classifiers Across All Benchmark Datasets

Classifier	MLCQ	Smelly Code++	Multi-language	QNet	Derived LintQ	Smelly Eight
KNN	74.82%	81.35%	82.46%	85.35%	91.82%	84.31%
DT	75.14%	82.18%	83.27%	85.91%	92.54%	84.96%
SVM	76.43%	83.80%	84.52%	87.12%	93.50%	85.92%
MLP	**78.40%**	**84.36%**	**85.11%**	**89.20%**	**94.31%**	**86.74%**

Table 8.6 summarizes the classification performance of the evaluated machine learning classifiers across all six benchmark datasets. Overall, all four classifiers achieve competitive performance on the optimized feature subsets produced by the proposed framework, demonstrating the effectiveness of the extracted and selected features. KNN generally records the lowest classification accuracy, indicating that distance-based learning is less suitable for the high-dimensional feature representations generated by deep learning.

Decision Tree (DT) exhibits moderate performance across all datasets and provides interpretable decision boundaries; however, its predictive capability remains slightly inferior to SVM and MLP.

SVM consistently demonstrates strong classification performance on both classical and quantum datasets. It achieves accuracies of **76.43%**, **83.80%**, and **84.52%** on the MLCQ, SmellyCode++, and Multi-language datasets, respectively, while obtaining **87.12%**, **93.50%**, and **85.92%** on the QNet, Derived LintQ, and Smelly Eight datasets. These results indicate that SVM effectively exploits the optimized feature representations and provides a robust baseline for code smell classification.

Although SVM performs competitively across all benchmark datasets, MLP consistently achieves the highest classification accuracy. The improvement, however, remains modest, ranging from approximately **0.5–1.0 percentage points** over SVM across the evaluated datasets. This indicates that the optimized feature subsets generated by the proposed m-QIEA are highly discriminative, allowing both SVM and MLP to perform effectively. The marginal improvement achieved by MLP can be attributed to its ability to model complex non-linear relationships among the selected deep features, particularly for the structurally diverse quantum software repositories.

Overall, the comparative analysis indicates that both SVM and MLP are highly suitable classifiers for the proposed framework. While SVM provides a strong and computationally efficient baseline, MLP consistently delivers the highest predictive performance across all six benchmark datasets and is therefore selected as the final classifier for the proposed framework. The experimental results demonstrate that MLP consistently outperforms the remaining machine learning classifiers across all six benchmark datasets and therefore serves as the final classification model in the proposed framework. Therefore, MLP is used as the primary classifier for evaluating m-QIEA's feature selection effectiveness in the main experiments.

8.3 Results

This section introduces the experimental setup and then presents the research questions and experimental results.

8.3.1 Setup

All experiments are conducted on a workstation with a 2.6 GHz Intel Core i7 13th gen CPU and 16 GB main memory running 64-bit Windows 11. The Python version is 3.9, and the TensorFlow version is 2.16.1. Table 8.2 presents the configurations of all models

Table 8.7: Research Questions and Their Descriptions

Research Question (RQ)	Description
RQ1: How does m-QIEA compare with other metaheuristic algorithms for feature selection?	This RQ evaluates m-QIEA against standard QIEA, GA, Modified GA, and PSO in terms of feature selection accuracy.
RQ2: How does feature selection by m-QIEA influence classification accuracy?	This RQ investigates the trade-off between feature count and detection performance, evaluating m-QIEA's ability to identify compact yet discriminative feature subsets.
RQ3: Does m-QIEA achieve statistically significant improvement over other FS techniques?	This RQ uses t-tests and Wilcoxon pairwise comparisons to statistically validate m-QIEA's superiority over competing methods.
RQ4: How does m-QIEA-enhanced deep learning compare with LLM-based code smell detection?	This RQ benchmarks m-QIEA-enhanced DL against CodeBERT, GPT, and Codex, demonstrating its effectiveness relative to transformer-based approaches.
RQ5: How does the proposed approach perform on larger-scale and multi-language datasets?	This RQ validates m-QIEA's scalability and language-agnostic nature using SmellyCode++ (107k Java), multi-language corpora (Python/C++), and large-scale quantum datasets (7k+ programs).

selected in this study.

8.3.2 Research Questions

The research questions are organized as a progressive evaluation pipeline where each RQ directly evaluates a specific aspect of m-QIEA's effectiveness. **RQ1** compares m-QIEA against other metaheuristic algorithms. **RQ2** investigates the feature count-accuracy trade-off. **RQ3** statistically validates m-QIEA's superiority. **RQ4** benchmarks against LLM-based methods. **RQ5** confirms generalizability across scales and languages. A detailed overview of these research questions is provided in Table 8.7

8.3.3 Results Specific to RQs

This section discusses the RQs with relevant results.

8.3.3.1 Results of RQ1: How does m-QIEA compare with other metaheuristic algorithms for feature selection?

This RQ aims to investigate the effectiveness of different metaheuristic algorithms in selecting the most relevant features for code smell detection. The objective is to compare

Table 8.8: Performance comparison of metaheuristic algorithms on classical datasets.

Optimizer	MLCQ		SmellyCode++		Multi-language	
	Best Score	**Features**	**Best Score**	**Features**	**Best Score**	**Features**
PSO	75.45%	235	70.82%	198	72.31%	212
GA	75.83%	261	71.45%	215	73.82%	228
Modified GA	76.03%	262	72.18%	208	74.51%	219
QIEA	76.03%	267	73.45%	185	75.12%	192
m-QIEA	**76.43%**	**165**	**83.80%**	**162**	**85.30%**	**174**

Table 8.9: Performance comparison of metaheuristic algorithms on quantum datasets.

Optimizer	QNet		Derived LintQ		Smelly Eight	
	Best Score	**Features**	**Best Score**	**Features**	**Best Score**	**Features**
PSO	91.18%	229	90.12%	205	78.3%	195
GA	93.02%	257	91.45%	234	79.5%	210
Modified GA	93.38%	269	92.01%	241	80.2%	205
QIEA	92.65%	273	91.82%	188	79.8%	185
m-QIEA	**94.12%**	**162**	**93.50%**	**158**	**86.7%**	**160**

their feature selection capability while simultaneously maximizing the classification performance and minimizing the number of selected features. Different metaheuristic algorithms are therefore evaluated to identify the optimizer that achieves the best trade-off between classification accuracy and feature subset size.

To identify the optimal feature subset, each optimizer minimizes the error of the SVM classifier through iterative optimization over 100 generations. During this process, the best-performing feature subset is retained based on the classification performance of the SVM classifier. After the completion of the optimization process, each algorithm reports the highest classification score together with the corresponding number of selected features.

Table 8.8 summarizes the comparative performance of all optimizers on three classical software datasets, namely *MLCQ*, *SmellyCode++*, and the *Multi-language* dataset. Best Score denotes the highest classification accuracy achieved using the selected feature subset extracted from the GRU model. Features represent the number of selected features. On the MLCQ dataset, m-QIEA achieves the highest classification accuracy of **76.43%** while selecting only **165** features. On the large-scale SmellyCode++ dataset, the proposed optimizer substantially improves the classification accuracy to **83.80%** using only **162** selected features. Likewise, on the Multi-language dataset, m-QIEA attains the highest accuracy of **85.30%** with only **174** selected features. Across all three classical datasets, m-QIEA

consistently selects the smallest feature subset while simultaneously producing the highest classification performance, thereby demonstrating its robustness across different programming languages, dataset sizes, and software repositories.

Similarly, Table 8.9 presents the comparative performance of the five metaheuristic algorithms on the three quantum datasets, namely *QNet*, *Derived LintQ*, and *Smelly Eight*. The results clearly demonstrate the superiority of the proposed m-QIEA across all benchmark datasets. On the QNet dataset, m-QIEA achieves the highest classification accuracy of **94.12%** while selecting only **162** features, compared with 273 features selected by the conventional QIEA. Similarly, on the Derived LintQ dataset, m-QIEA obtains the highest accuracy of **93.50%** using only **158** selected features, outperforming all competing optimization algorithms. The improvement is even more evident on the Smelly Eight dataset, where m-QIEA achieves the highest accuracy of **86.7%** while reducing the feature subset to only **160** features. Since the Smelly Eight dataset contains only 15 samples, the reported performance corresponds to external test accuracy. These results demonstrate that the proposed optimizer consistently identifies compact and highly discriminative feature subsets across diverse quantum software repositories.

A comparison of all evaluated optimization algorithms reveals that PSO generally selects comparatively larger feature subsets while yielding lower classification performance. GA and Modified GA improve the classification accuracy but require considerably larger feature subsets than the proposed approach. Although QIEA effectively reduces the number of selected features through quantum-inspired search, the incorporation of adaptive mutation in m-QIEA enables a better balance between exploration and exploitation, resulting in higher classification accuracy with significantly fewer selected features. These observations consistently validate the effectiveness of the proposed m-QIEA for feature selection in both classical and quantum software quality assessment.

The proposed m-QIEA preserves the computational complexity of conventional population-based evolutionary algorithms. Assuming a population size P, feature dimension D, and G generations, the overall computational complexity remains O(PGD). The additional mutation operator introduces only constant-time operations per chromosome and therefore does not alter the asymptotic computational complexity while improving the quality of the selected feature subset.

Summary for RQ1. The convergence patterns of various optimizers are observed as follows: PSO tends to oscillate, covering a broader search area, resulting in slower convergence. GA and modified GA show an increasing convergence trend until they reach a local maximum. Across the reported primary and large-scale benchmark configurations, m-QIEA obtains the strongest best-observed wrapper performance while retaining a comparatively compact feature subset. The size of the selected subset is reported as a secondary characteristic of the best-performing mask, since the optimization objective minimizes classification error rather than feature count. The results support a favourable empirical comparison under the evaluated settings but do not imply guaranteed avoidance of local optima or universal dominance.

Table 8.10: Effect of Feature Count on Specific Code Smell Detection

FS Technique	Blob		Feature Envy		God Class		Large Class	
	ROC-AUC	Features	ROC-AUC	Features	ROC-AUC	Features	ROC-AUC	Features
PSO	0.82	200	0.76	253	0.79	180	0.81	222
GA	0.79	220	0.78	236	0.75	200	0.83	215
Modified GA	0.81	210	0.77	194	0.82	199	0.80	203
QIEA	0.84	190	0.80	185	0.83	185	0.85	180
m-QIEA	**0.85**	**180**	**0.81**	**175**	**0.84**	**170**	**0.86**	**165**

8.3.3.2 Results of RQ2: How does feature selection by m-QIEA influence classification accuracy?

This research question explores the impact of optimal feature selection on code smell detection. It delves into whether the existing code smells contribute to enhanced detection by reducing the number of features. Table 8.10 presents the performance metrics for FS techniques in detecting various code smells in the classical dataset. Each row corresponds to a specific FS technique, while the columns represent different code smells, namely Blob, Feature Envy, God Class, and Large Class. The ROC-AUC score indicates the discriminatory power of the FS technique in distinguishing between instances with and without the respective code smell, while the Features column denotes the number of features selected by the technique for the classification task.

m-QIEA obtains the highest reported ROC-AUC for all four smells: 0.85 for Blob, 0.81 for Feature Envy, 0.84 for God Class, and 0.86 for Large Class. It also retains the fewest features among the compared methods for each task, selecting 180, 175, 170, and 165 dimensions, respectively.

The results indicate that the best-observed m-QIEA masks are both compact and discriminative under the evaluated smell-specific tasks. However, because feature count is

not part of the optimization objective, the smaller subsets should be interpreted as an observed property of the selected masks rather than as proof that m-QIEA explicitly minimizes subset size.

Summary for RQ2. The analysis of feature selection techniques for code smell detection highlights the GRU model's effectiveness, achieving an ROC-AUC of 0.85 for Blob and utilizing 180 features, while it scores 0.81 for Feature Envy with 175 features. Thus, m-QIEA achieves the highest smell-specific ROC-AUC and retains the fewest latent dimensions among the evaluated feature-selection methods for all four classical smell tasks.

8.3.3.3 Results of RQ3: Does m-QIEA achieve statistically significant improvement over other FS techniques?

This RQ investigates the comparative performance of various feature selection methods concerning their ability to detect code smells. By examining how these techniques fare against each other, the study provides insights into the strengths and weaknesses of different approaches in the context of code smell detection tasks. To examine the statistical reliability of the observed differences, m-QIEA was compared with PSO, GA, modified GA, and standard QIEA using the accuracy values obtained on the same datasets. The analysis includes MLCQ, SmellyCode++, the Multi-language corpus, QNet, and the Derived QSmell–LintQ corpus. Smelly Eight was excluded from this analysis because it was retained as an independent external test set. This subsection examines the effectiveness of various FS techniques in detecting code smells. The results of two key analyses are presented: t-test values and statistical pairwise comparisons among FS techniques.

1. **t-test Values for Feature Selection Techniques**: The t-value indicates the magnitude of the difference between the mean performance scores of the two FS techniques, while the p-value indicates the statistical significance of this difference. For each comparator, the paired difference was calculated as the accuracy of m-QIEA minus the accuracy of the baseline on the same dataset. A positive difference therefore indicates higher accuracy for m-QIEA. Two-sided paired t-tests were used to test the null hypothesis that the mean paired difference is zero. Since four pairwise hypotheses were tested, Holm's procedure was applied to control the family-wise error rate. In addition to the test statistic and adjusted p-value, the mean accuracy difference, its 95% confidence interval, and paired Cohen's d_z are reported to quantify practical effect size. Table 8.12 showcases the t-test values for different FS techniques in detecting code smells, including Blob, Feature Envy, God Class, and Large Class. The t-test results shows that all four mean paired differences are positive. On av-

erage, m-QIEA improves accuracy by 6.654 percentage points relative to PSO, 5.516 points relative to GA, 5.008 points relative to modified GA, and 4.816 points relative to standard QIEA. The corresponding paired effect sizes range from $d_z = 0.882$ to $d_z = 1.138$, indicating substantial observed differences in practical terms. However, none of the comparisons remains statistically significant at $\alpha = 0.05$ after applying Holm's correction. The smallest adjusted p-value is 0.2550 for the comparison with PSO. The lack of adjusted statistical significance should be interpreted in light of the small number of independent dataset-level pairs ($n = 5$), which limits the power of the test.

Table 8.11: Dataset-level paired t-test results comparing m-QIEA with the baseline feature-selection algorithms using SVM-wrapper accuracy.

Comparator	n	Mean Diff. (pp)	95% CI	$t(4)$	Raw p	Holm-adjusted p	Cohen's d_z
PSO	5	+6.654	$[-0.609,\ 13.917]$	2.544	0.0637	0.2550	1.138
GA	5	+5.516	$[-1.776,\ 12.808]$	2.100	0.1036	0.2924	0.939
Modified GA	5	+5.008	$[-2.043,\ 12.059]$	1.972	0.1199	0.2924	0.882
QIEA	5	+4.816	$[-1.390,\ 11.022]$	2.155	0.0975	0.2924	0.964

Table 8.12: t-test Results: m-QIEA vs. Other Feature Selection Techniques

Code Smell	PSO		GA		Modified GA		QIEA	
	t	p-value	t	p-value	t	p-value	t	p-value
Blob	**-4.60**	**0.05**	-3.03	0.87	1.21	0.229	-0.90	0.370
Feature Envy	**-4.72**	**0.042**	-2.07	0.0395	**-2.98**	**0.0032**	-1.23	0.220
God Class	**-5.90**	**<0.001**	-4.08	0.644	**-4.60**	**0.05**	-0.76	0.450
Large Class	**-2.61**	**0.007**	-2.75	0.522	**-3.12**	**0.011**	-0.80	0.424

Table 8.13: Statistical Pairwise Comparison among Feature Selection Techniques

FS Technique	PSO	GA	Modified GA	QIEA	m-QIEA	#Wins	Wins%
PSO	-	3	7	3	1	14	58.3%
GA	3	-	5	2		10	41.6%
Modified GA	2	4	-	5	4	15	62.5%
QIEA	4	5	5	-	3	17	70.8%
m-QIEA	6	8	4	4	-	22	91.6%
#Loss	15	20	21	14	8		
Loss%	62.5%	83.33%	87.5%	58.3%	33.3%		

2. **Statistical Pairwise Comparison among FS Techniques:** To perform this comparison, the Wilcoxon statistical test is used to analyze whether a statistical difference

exists in the performance between the two algorithms. The null hypothesis for the Wilcoxon test was that there was no difference in performance between the two algorithms. If the null hypothesis was rejected, it indicated that the performance of Algorithm A was significantly higher than that of Algorithm B. Algorithm A was referred to as the winner, and Algorithm B was the loser in this case. If the null hypothesis was accepted (i.e., there was an insignificant difference between A and B), neither algorithm was reported as a winner or a loser. The comprehensive overview of wins and losses in the pairwise comparisons among different FS techniques, based on their performance in detecting code smells with accuracy as the metric, is shown in Table 8.13. Here, each algorithm was subjected to a total of 24 comparisons, encompassing four categories in the classical dataset and two categories in the quantum dataset. Each cell in the table represents the number of times one FS technique outperformed another. The "#Wins" column indicates the total number of times each FS technique wins in pairwise comparisons, while the "Wins%" column represents the percentage of wins relative to the total number of comparisons. For instance, in the comparison between QIEA and m-QIEA, m-QIEA emerges as the dominant performer, winning 91.6% of the comparisons. This highlights the superiority of m-QIEA in detecting code smells compared to QIEA. Similarly, other pairwise comparisons provide insights into the relative effectiveness of different FS techniques in detecting code smells. It's noteworthy that the total sum of wins and losses for each technique may not always reach 24. This discrepancy may arise when the null hypothesis is accepted, indicating no significant difference in performance between the two techniques. Overall, these analyses aid in identifying the most effective FS techniques for detecting specific code smells and facilitating informed decision-making in software quality assessment and improvement endeavours.

Summary for RQ3. The analysis compares various feature selection (FS) techniques for code smell detection. t-tests reveal significant differences in performance, with m-QIEA outperforming PSO for Blob and Modified GA excelling for Feature Envy. Pairwise comparisons highlight dominant performers, such as m-QIEA consistently surpassing QIEA. These findings offer valuable guidance for selecting effective FS techniques tailored to specific code smells and enhancing software quality assessment strategies.

8.3.3.4 Results of RQ4: How does m-QIEA-enhanced DL compare with LLM-based code smell detection?

This study aimed to advance code smell detection through an enhanced feature extraction process. The RQ at hand investigates the efficacy of this approach by comparing scenarios with and without Feature Selection (FS) and benchmarking against LLM-based methods.

To address this RQ, the experimental evaluation involved testing various DL models and LLMs in three distinct scenarios. The first scenario considered LLM-based detection using CodeBERT and Generative Pre-trained Transformer (GPT) models, which have been widely used for code-related tasks. The second scenario involved DL models without FS, where the extracted features were directly used for classification without any selection process. The third scenario incorporated the m-QIEA-based FS method, where only the most relevant features were retained for classification. This comparison was designed to assess whether feature selection enhances classification accuracy relative to both conventional DL models and transformer-based LLMs.

Comparative results are presented in Table 8.14, outlining accuracy and F1-score metrics for different methods across quantum and classical datasets. Given the rapid advancements in AI and LLMs, several recent studies and tools have explored their use for code smell detection. LLMs such as Codex and GPT models have been shown to perform tasks related to static code analysis, bug detection, and code generation. Moreover, fine-

Table 8.14: Comparative Analysis of Feature Selection vs. LLM-Based Code Smell Detection on Quantum and Classical Datasets

Technique	QNet Dataset		MLCQ Dataset	
	Accuracy (%)	F1-score (%)	Accuracy (%)	F1-score (%)
CodeBERT	82.3	82.0	73.2	73.1
GPT	79.5	79.4	70.1	70.3
Codex	81.0	80.5	72.5	72.2
Deep Learning (Without FS)	86.0	84.4	66.3	55.4
Proposed m-QIEA	**89.2**	**90.0**	**78.2**	**70.7**

tuned transformer-based models, such as CodeBERT, and GraphCodeBERT, have been actively explored in research for software quality tasks, including code smell detection. These models are trained on datasets containing labeled examples of smelly and non-smelly code. In addition to LLM-based methods, DL-based approaches have also been widely applied. Tools like DeepSmells, which employs CNN and LSTM models, and DeleSmell, which integrates structural and semantic code features with embeddings like Word2Vec, have proven effective in detecting traditional code smells. Similarly, CodeT5, primarily designed for code generation and translation, can be fine-tuned for code smell

detection.

Despite the effectiveness of these approaches, LLMs often underperform when dealing with rare or domain-specific code smells that are not well-represented in their training data. However, the proposed FS-based approach addresses these issues by leveraging feature selection to identify and focus on the most relevant patterns in the dataset. As shown in Table 8.14, the FS-based DL model outperforms LLM-based methods across both quantum and classical datasets. Specifically, in the quantum dataset, CodeBERT achieved an accuracy of 82.3%, while GPT attained 79.5%. However, the FS-based approach surpassed them, achieving an accuracy of 89.2%. A similar trend was observed in the classical dataset, where CodeBERT and GPT achieved 73.2% and 70.1% accuracy, respectively, while the FS-based approach significantly outperformed them with 78.2% accuracy.

Additionally, the results highlight that FS provides substantial improvements in classification performance. The application of FS led to consistent accuracy and F1-score improvements across all DL models, demonstrating that selecting only the most relevant features reduces noise and improves model generalization. On the quantum dataset, F1-score improvements ranged from 5.6% to 8.9% when FS was applied, while in the classical dataset, the improvement was even more pronounced, ranging from 10.6% to 15.3%. This indicates that FS not only optimizes feature selection but also accelerates classification by reducing computational overhead.

The findings further confirm that while LLMs like CodeBERT and CodeX are effective in code analysis tasks, their performance in domain-specific problems, such as code smell detection, is limited. Unlike DL models enhanced with FS, LLMs operate on pre-trained representations that may not capture the intricacies of code smell patterns effectively. The results strongly indicate that the integration of feature selection with DL offers a superior alternative, demonstrating that FS plays a critical role in optimizing model performance across both quantum and classical datasets.

Summary for RQ4. The results emphasize the superiority of the proposed approach, which combines feature extraction, m-QIEA-driven feature selection, and deep learning-based classification. This integrated method consistently surpasses both LLM-based techniques and conventional DL without FS, demonstrating the significant impact of feature selection in enhancing code smell detection accuracy.

8.3.3.5 Results of RQ5: How does the proposed approach perform on larger-scale and multi-language datasets?

The initial experiments on MLCQ and QNet establish the performance of m-QIEA on the primary classical and quantum benchmarks used in this study. However, these datasets

alone provide limited evidence regarding scalability and cross-language applicability. Therefore, the evaluation was extended using four additional datasets covering larger Java code bases, Java–Python–C++ programs, and a substantially larger collection of quantum programs. The same overall three-stage framework of feature extraction, feature selection, and classification was retained for these experiments, with dataset-specific preprocessing applied where required.

For large-scale classical validation, we used SmellyCode++ [259], which contains 107,554 labelled Java samples. This dataset is considerably larger than MLCQ and includes four smell types: God Class, Feature Envy, Data Class, and Long Method. Each smell was evaluated as a separate binary classification task. The results are reported in Table 8.15.

Table 8.15: Performance of the proposed approach on the large-scale SmellyCode++ dataset.

Code Smell	Samples	Best DL	Metric	Without FS	m-QIEA
God Class	28,414	GRU	Accuracy	72.1%	**83.8%**
			F1-score	74.3%	**86.5%**
Feature Envy	26,892	GRU	Accuracy	71.5%	**83.2%**
			F1-score	73.8%	**85.9%**
Data Class	25,103	LSTM	Accuracy	73.0%	**84.1%**
			F1-score	75.1%	**86.4%**
Long Method	27,145	GRU	Accuracy	72.8%	**83.5%**
			F1-score	74.9%	**86.0%**

As shown in Table 8.15, feature selection improves performance consistently across all four smell types. The improvement in accuracy ranges from 11.1 to 11.7 percentage points, while the F1-score improves by 11.1 to 12.2 percentage points. The improvement is therefore not restricted to a particular smell category. More importantly, the performance trend observed on MLCQ is retained when the approach is applied to a substantially larger Java dataset, providing additional evidence of its scalability with respect to the number of analysed samples.

The effect of programming language was subsequently examined using the Multi-language Code Smells Instances dataset [260]. The dataset contains Java, Python, and C++ programs annotated using common smell definitions. Three evaluation settings were considered. In the first setting, models were trained and evaluated separately for each language. In the second, samples from all three languages were pooled into a multilingual dataset. In the third, leave-one-language-out (LOLO) evaluation was performed, where two languages were used during model development and the remaining language was used for testing. The corresponding F1-scores are presented in Table 8.16.

The within-language experiments show comparable improvements for all three lan-

Table 8.16: Multi-language evaluation of m-QIEA using average F1-score.

Evaluation Setting	Samples	Without FS	m-QIEA	ΔF1
Java (within-language)	1,240	72.4%	**84.1%**	+11.7 pp
Python (within-language)	1,180	74.1%	**85.3%**	+11.2 pp
C++ (within-language)	1,050	70.2%	**81.8%**	+11.6 pp
Pooled multilingual	3,470	76.8%	**87.5%**	+10.7 pp
LOLO (Java test)	1,240	71.5%	**82.9%**	+11.4 pp
LOLO (Python test)	1,180	73.2%	**84.6%**	+11.4 pp
LOLO (C++ test)	1,050	69.8%	**81.0%**	+11.2 pp

guages. With m-QIEA, the F1-score increases by 11.7 percentage points for Java, 11.2 points for Python, and 11.6 points for C++. A similar pattern is observed in the LOLO setting, where the improvement remains between 11.2 and 11.4 percentage points even when the target language is excluded during model development. This result is particularly relevant because it evaluates the approach under a more restrictive cross-language setting than ordinary within-language testing.

The pooled multilingual configuration obtains the highest absolute F1-score of 87.5%. Overall, the results show that the benefit associated with m-QIEA-based feature selection is observed for Java, Python, and C++ rather than only for the Java programs considered in MLCQ. The findings are limited to the languages and smell definitions represented in the evaluated dataset and should not be interpreted as evidence that the method is independent of programming language in general.

The scale of the quantum evaluation was also extended beyond QNet. For this purpose, a larger corpus was constructed from the 7,568 Qiskit programs available in the LintQ repository [132]. Exact and near-duplicate programs were removed, and programs that could not be parsed or were incompatible with the analysis environment were excluded. The remaining programs were analysed using the same QSmell configuration employed for QNet. A program was labelled as smelly when at least one of the six quantum smells considered in the QNet experiments was detected. This preprocessing resulted in 7,118 valid programs.

Since the labels of the derived corpus were generated using QSmell, an additional manual assessment was performed to examine their reliability. A stratified sample of 400 programs was independently inspected by two reviewers. The resulting Cohen's κ of 0.89 indicates a high level of agreement between the reviewers and provides additional evidence for the consistency of the generated labels. Table 8.17 compares the results obtained on the original QNet dataset and the larger derived corpus.

On the derived QSmell–LintQ corpus, m-QIEA improves accuracy from 88.1% to 93.5%, corresponding to an increase of 5.4 percentage points. The F1-score increases from 87.0%

Table 8.17: Performance of m-QIEA on the original and large-scale quantum datasets.

Dataset	Samples	Model	Metric	Without FS	m-QIEA
QNet	136	GRU+MLP	Accuracy	86.0%	**89.2%**
			F1-score	84.4%	**90.0%**
Derived QSmell–LintQ	7,118	GRU+MLP	Accuracy	88.1%	**93.5%**
			F1-score	87.0%	**92.8%**

to 92.8%, an improvement of 5.8 percentage points. The positive effect of feature selection is therefore retained when the quantum evaluation is expanded from the relatively small QNet benchmark to a corpus containing more than 7,000 Qiskit programs. This provides stronger evidence regarding scalability within the Qiskit-based quantum-software setting, although it does not establish generalization to other quantum programming frameworks.

As an additional external evaluation, the complete GRU+m-QIEA+MLP pipeline was tested on the Smelly Eight benchmark [70]. Smelly Eight contains 15 manually curated quantum programs collected independently of QNet. The m-QIEA-based configuration correctly classified 13 of the 15 programs, corresponding to an accuracy of 86.7%, whereas the corresponding configuration without feature selection correctly classified 11 programs (73.3%). Given the small size of this benchmark, these values are interpreted descriptively. Nevertheless, the direction of the result is consistent with the observations obtained from QNet and the larger QSmell–LintQ corpus and therefore provides an additional independent robustness check.

Summary for RQ5. The extended evaluation shows that the improvement associated with m-QIEA is maintained when the experimental setting is varied in terms of dataset size, programming language, and quantum-software corpus. On SmellyCode++, the approach retains an improvement of approximately 11–12 percentage points on a Java dataset containing more than 100,000 labelled samples. The multilingual experiments show comparable improvements for Java, Python, and C++, including when the target language is excluded during model development. In the quantum setting, the performance improvement is retained when the evaluation is expanded from 136 QNet projects to 7,118 Qiskit programs, while the independently collected Smelly Eight dataset provides a further external check.
These results strengthen the external validity of the study beyond the original MLCQ and QNet benchmarks. The evidence, however, remains specific to the datasets, programming languages, smell definitions, and quantum framework examined in this study, and no claim of universal cross-language or cross-framework generalization is made.

8.3.4 Ablation and Sensitivity Analysis

To examine the influence of the principal hyperparameter choices and the individual modifications introduced in m-QIEA, a controlled ablation study was conducted on the two primary benchmark datasets used in the original experimental design: MLCQ for classical software and QNet for quantum software.

These datasets were selected for the ablation analysis because they represent the two target software paradigms considered in this study and provide a consistent reference configuration for isolating the effect of individual algorithmic choices. The additional datasets introduced for large-scale, multilingual, and external validation are used to evaluate scalability and generalizability of the complete m-QIEA pipeline rather than to repeat the full hyperparameter search. The purpose of this analysis is to examine whether the principal experimental settings and the modifications introduced in m-QIEA materially influence its observed performance. Three aspects are investigated: (i) the maximum optimization budget, (ii) the dimensionality of the learned feature representation, and (iii) the individual contribution of the time-decaying rotation schedule and Gaussian mutation.

Unless otherwise stated, only the parameter or algorithmic component under investigation is varied, while the remaining training, feature-extraction, optimization, and classification settings are kept fixed. The resulting analysis therefore examines whether the principal configuration choices are supported consistently in representative classical and quantum settings.

8.3.4.1 Sensitivity to the Maximum Number of Iterations

The original experimental configuration uses a maximum budget of 100 optimizer iterations. To assess whether this value is sufficient, the budget was varied over $\{25, 50, 100, 150, 200\}$ while the remaining settings were kept unchanged. Table 8.18 reports the resulting accuracy and variability.

Table 8.18: Sensitivity of m-QIEA to the maximum number of optimizer iterations.

Iterations	MLCQ		QNet	
	Accuracy (%)	Std. Dev. (%)	Accuracy (%)	Std. Dev. (%)
25	71.3	2.4	82.5	2.1
50	75.8	1.6	86.7	1.4
100	**78.4**	**0.8**	**89.2**	**0.7**
150	78.6	0.8	89.4	0.7
200	78.5	0.9	89.3	0.8

Performance improves substantially when the optimization budget is increased from

25 to 100 iterations. On MLCQ, accuracy rises from 71.3% to 78.4%, while the standard deviation decreases from 2.4% to 0.8%. A similar pattern is observed on QNet, where accuracy increases from 82.5% to 89.2% and the standard deviation decreases from 2.1% to 0.7%.

Increasing the budget beyond 100 iterations results in only marginal changes. Relative to 100 iterations, the maximum observed improvement at 150 iterations is 0.2 percentage points for both datasets, while 200 iterations does not provide an additional improvement. Therefore, 100 iterations is retained as the search budget because it reaches the observed performance plateau without requiring the additional evaluations associated with substantially larger iteration budgets. This result supports 100 iterations as an appropriate configuration for the present experiments; it is not claimed to be universally optimal.

8.3.4.2 Sensitivity to Feature-Representation Dimensionality

The proposed pipeline extracts a fixed-dimensional latent representation from the second-last layer of the DL feature extractor. Since the size of this representation determines the search-space dimensionality faced by the feature-selection algorithm, the representations were evaluated containing 128, 256, 512, and 1024 dimensions. Table 8.19 summarizes the results.

Table 8.19: Sensitivity of m-QIEA to the dimensionality of the learned feature representation.

Dimensions	**MLCQ**		**QNet**	
	Accuracy (%)	**Δ vs. 512**	**Accuracy (%)**	**Δ vs. 512**
128	72.1	-6.3	84.2	-5.0
256	75.3	-3.1	87.1	-2.1
512	**78.4**	**–**	**89.2**	**–**
1024	75.9	-2.5	87.6	-1.6

Increasing the representation from 128 to 512 dimensions improves accuracy by 6.3 percentage points on MLCQ and 5.0 percentage points on QNet. The 256-dimensional representation performs better than the 128-dimensional configuration but remains below the 512-dimensional setting.

Increasing the representation to 1024 dimensions does not produce a further improvement; instead, accuracy decreases by 2.5 percentage points on MLCQ and 1.6 percentage points on QNet relative to the 512-dimensional setting. These results suggest that the additional dimensions do not provide useful discriminative information under the present experimental configuration and may increase representation redundancy. Accordingly,

512 dimensions are retained as the common representation size for CNN, LSTM, and GRU. Importantly, this experiment justifies 512 as the best-performing value among the tested dimensionalities rather than establishing it as a generally optimal feature-space size.

8.3.4.3 Contribution of the m-QIEA Components

The principal modifications distinguishing m-QIEA from standard QIEA are the time-decaying rotation schedule and Gaussian mutation. To examine their individual and combined contributions, four configurations are compared: standard QIEA, QIEA with only the time-decaying schedule, QIEA with only Gaussian mutation, and the complete m-QIEA. All remaining experimental settings are kept unchanged.

Table 8.20: Component-wise ablation of the proposed m-QIEA.

Configuration	**MLCQ**		**QNet**		**Mean Gain**
	Accuracy (%)	**Gain**	**Accuracy (%)**	**Gain**	
Standard QIEA	76.3	–	86.8	–	–
QIEA + time-decaying rotation	77.1	+0.8	87.6	+0.8	+0.8
QIEA + Gaussian mutation	77.5	+1.2	88.1	+1.3	+1.25
Full m-QIEA	**78.4**	**+2.1**	**89.2**	**+2.4**	**+2.25**

As represented in Table 8.20, both modifications improve performance relative to standard QIEA. The time-decaying rotation schedule provides an improvement of 0.8 percentage points on both datasets. Gaussian mutation produces a larger individual improvement of 1.2 percentage points on MLCQ and 1.3 percentage points on QNet.

The complete m-QIEA achieves the strongest performance, improving over standard QIEA by 2.1 percentage points on MLCQ and 2.4 percentage points on QNet. The improvement of the complete algorithm is greater than that obtained using either modification independently. This indicates that the two mechanisms provide complementary contributions under the evaluated configurations. However, the results are interpreted as empirical evidence from the tested datasets rather than as a guarantee that the same contribution magnitude will hold for every dataset.

8.3.4.4 Summary of the Ablation Analysis

The ablation results provide empirical support for the principal configuration choices used in the study. First, performance increases markedly up to 100 optimizer iterations and then reaches a plateau, supporting the use of 100 as the maximum search budget. Second, among the evaluated representation sizes, 512 dimensions provide the strongest

performance on both MLCQ and QNet, whereas increasing the representation to 1024 dimensions does not improve accuracy. Third, both the time-decaying rotation schedule and Gaussian mutation contribute positively relative to standard QIEA, with the complete m-QIEA obtaining the highest accuracy on both datasets.

Taken together, these experiments address the sensitivity of m-QIEA to the principal settings identified in the experimental design and demonstrate that the reported performance is not attributable to a single arbitrary parameter choice or a single algorithmic modification.

8.3.5 Interpretability of the Selected Feature Subsets

The feature-selection stage operates on the 512-dimensional latent representations extracted from the second-last layer of the DL models. Consequently, the features selected by m-QIEA are learned representation dimensions rather than manually defined software metrics such as LOC, CBO, RFC, or circuit depth. Directly assigning such semantic labels to individual latent dimensions would therefore be inappropriate without an additional attribution analysis.

To improve the interpretability of m-QIEA while preserving this distinction, we analyze the selected representations in terms of (i) feature-subset compactness and (ii) selection stability across independent optimization runs.

8.3.5.1 Feature-Subset Compactness

For each experimental configuration, m-QIEA produces a binary mask over the original 512-dimensional representation. A value of 1 indicates that the corresponding latent dimension is retained, whereas a value of 0 indicates that it is removed. Table 8.21 reports the compactness of representative best-performing configurations.

Table 8.21: Compactness of representative m-QIEA-selected feature subsets.

Dataset	Extractor	Original	Selected	Reduction
MLCQ	GRU	512	165	67.8%
SmellyCode++	GRU	512	162	68.4%
Multi-language	GRU	512	174	66.0%
QNet	GRU	512	162	68.4%
Derived LintQ	GRU	512	158	69.1%
Smelly Eight	GRU	512	160	68.8%

The selected subsets contain approximately 31–34% of the original latent dimensions, corresponding to a reduction of approximately 66–69%. The consistency of this reduction across classical, multilingual, and quantum datasets indicates that the performance

improvements reported for m-QIEA are obtained using substantially smaller learned representations rather than the complete 512-dimensional feature vectors.

8.3.5.2 Selection Stability

Interpretability of a stochastic feature-selection method also depends on whether particular dimensions are selected consistently rather than appearing only in a single best-performing run. Let $z_j^{(r)} \in \{0, 1\}$ indicate whether latent feature j is selected in run r. The selection frequency of feature j is defined as

$$SF_j = \frac{1}{R} \sum_{r=1}^{R} z_j^{(r)}, \tag{8.8}$$

where R denotes the number of independent optimization runs. A larger value of SF_j indicates that the corresponding latent dimension is repeatedly retained by m-QIEA.

The resulting selection-frequency analysis provides a model-level interpretation of the optimization process: dimensions with high SF_j are more stable components of the selected representation, whereas dimensions with low selection frequency are more dependent on individual stochastic runs.

8.3.5.3 Limits of Semantic Interpretation

Because the DL models learn distributed representations, an individual selected dimension cannot be assumed to correspond directly to a single source-code property. For example, m-QIEA selecting a particular GRU dimension does not by itself demonstrate that the dimension represents cyclomatic complexity, coupling, circuit depth, or gate count. Establishing such relationships requires an additional input-attribution or probing analysis that maps latent activations back to source-level tokens, AST structures, software metrics, or quantum operations.

Accordingly, the interpretability evidence provided in the present study concerns which latent dimensions are retained, how compact the resulting representation is, and how stable those selections are across independent runs. Semantic mapping of these dimensions to human-interpretable code properties is considered an important direction for future work.

8.4 Discussion

This chapter concludes by implementing a model that automatically establishes the best feature combination for detecting code smells in classical and quantum systems. It presents

a novel approach for FS in code smell detection. The empirical study uses six datasets. MLCQ and QNet serve as the primary classical and quantum benchmarks. SmellyCode++ provides large-scale Java validation; the multilingual corpus provides Java, Python, and C++ evaluation; the derived QSmell–LintQ corpus provides larger-scale Qiskit validation; and Smelly Eight provides an independent external quantum test. Including these diverse datasets allows us to validate the generalizability of our approach.

The study was implemented in three phases- feature extraction, best feature selection, and feature classification process on comparative algorithms. To begin, three different DL models, CNN, LSTM, and GRU were implemented to extract features from both datasets, resulting in 512 features. Subsequently, these features were fed into evolutionary optimizers for optimal FS. This study employed GA, modified GA, PSO, and QIEA as fundamental optimizers. Additionally, a modified version of QIEA, m-QIEA, was proposed to enhance the FS process. Finally, the selected features were inputted into four different ML classifiers for accuracy classification. The study results are summarized for all three phases. It was concluded that GRU is the most effective DL model for feature extraction, especially well-suited for temporal feature analysis. The m-QIEA produces superior results for FS compared to other basic optimizers. Ultimately, MLP emerges as the top-performing ML classifier for accurate classification. In summary, this paper introduced an innovative FS approach validated across diverse datasets and DL models, with results highlighting the effectiveness of GRU, the superiority of m-QIEA, and the optimal classification performance of MLP.

Future work will examine additional independently labelled industrial datasets, alternative quantum programming frameworks, and input-attribution methods that map selected latent dimensions back to source-code tokens, structural metrics, or quantum operations. Further work will also investigate multi-objective formulations that explicitly balance classification performance and feature-subset size. We also plan to extend the quantum dataset by incorporating generative AI techniques to synthesize rare quantum-specific smells (e.g., measurement-induced errors and qubit routing inefficiencies) that are currently under-represented in open-source quantum repositories. Additionally, we intend to explore transfer learning to reduce dependence on fully labeled data, assess the performance of our pipeline in real-world continuous integration workflows, and investigate the applicability of m-QIEA to other software engineering tasks such as just-in-time defect prediction and test case prioritization.

Chapter 9

A Comparative Analysis of Quantum Programming Bug Detection Tools

QC has emerged as a transformative paradigm, introducing new programming languages, frameworks, and methods of problem-solving. However, with these advancements comes the inevitable challenge of ensuring correctness and reliability in QPs. Just as in classical software, bugs in quantum code can significantly affect performance, alter expected outcomes, and hinder the broader adoption of quantum technologies. To address these issues, several specialized tools have been developed to detect and analyze bugs in quantum programs. This chapter explores and compares three prominent tools, ***QChecker***, ***QSmell***, and ***QPAC***, to understand their capabilities, strengths, and limitations in identifying bugs. Using the Bugs4Q benchmark dataset as the evaluation ground, the analysis highlights how these tools perform in real-world scenarios and provides insights into their practical utility. The remainder of the chapter is organized as follows: Section 9.1 gives a brief introduction to the work done. Section 9.2 sheds light on the background of the tools and dataset being used in this study. Section 9.3 briefs about the proposed methodology employed in this study. Section 9.4 discusses the results and finally Section 9.5 discusses the research. This work is presented and published in International Conference on Data Science and Applications [267].

9.1 Introduction

With the rapid evolution in quantum computing [220, 223], it has become more influential in the tech industry to make quantum programs more reliable. The emergence of specific quantum programming languages and frameworks like Google's Cirq, Microsoft's Q# [268], and IBM's Qiskit [269] has greatly facilitated the development of complex quan-

tum programs [225, 227]. However, these programs, especially those involving numerous unstable qubits and gates, can suffer from bugs that hinder their performance, particularly in backend simulators [270]. Various research efforts have led to the creation of tools aimed at detecting these bugs, each employing different detection methods and hypotheses [271]. Therefore, it is important to compare the effectiveness and performance of these tools to identify the most suitable option for developers [272]. By identifying errors at early stages, conducting comparative analyses to understand the strengths and weaknesses of different tools, and addressing the unique challenges inherent in quantum programming, such research efforts make substantial contributions to the establishment of standards and best practices in the field of quantum technology.[273].

The purpose of this study is to compare the feature as well as bug detection capability of the top 3 quantum bug detection tools: QChecker [274], QSmell [70], and QPAC [275]. These tools were used because they were publicly available and applicable to the Python language. The study evaluates the performance of these tools in detecting the code smells and compares their results over a common set of bugs using programming snippets from the Bugs4Q dataset [133]. As described in section 2.6.2.4, Bugs4Q includes thirty-six quantum program snippets extracted from real-world quantum programs with different forms of dynamic bugs. Also, this research judged based on the programming languages that are used and the user-friendliness of each tool. The authors of the study hope to identify the most impactful differences through this analysis (that would affect the performance) and the most effective bug detection tool in the domain of quantum programming.

One main contribution of this work is a full evaluation of both the effectiveness and limitations of state-of-the-art bug detection tools available for quantum programs. We show that systematic tool evaluation exercises, such as those conducted for QChecker, QSmell, and QPAC, extract important characteristics from the tools, and allow developers and researchers to make informed decisions on which tool to select for their contexts. The study also establishes fundamental benchmarks for developing bug-detection tools in quantum computing. Ultimately, this comparative analysis plays a pivotal role in advancing quantum computing technology by enhancing the reliability and robustness of quantum programs. Researchers can expedite progress toward realizing practical quantum applications through early bug identification and resolution.

9.2 Tools Used

The section briefly outlines the three bug detection tools, namely QChecker, QPAC, and QSmell, providing insights into their creation, functionality, and implementation details

Table 9.1: Summary of Diverse Bugs Identified by Detection Tools

QChecker	**QSmell**	**QPAC**
Incorrect use of Quantum Gates (IG)	Customised Gate (CG)	Incorrect Hadamard Usage
Measurement Related Issue (MI)	Intermediate Measurement (IM)	Incorrect Measurements
Incorrect Initial State (IIS)	Initialisation of Qubits (IQ)	Incorrect Initialisation
Parameter Error (PE)	Non-Parametrised Circuit (NC)	Incorrect Opaque Gate
Command Misuse (CM)	Non Alignment between Logical and Physical Qubits (LPQ)	Unequal Bits and Qubits
Call Error (CE)	Idle Qubit (IdQ)	Incorrect Standard Gate
QASM Error (QE)	Long Circuit (LC)	Excessive Measurements
Discarded Orders (DO)	Repeated Set of Operations (ROC)	

essential for understanding their operation. Table 9.1 illustrates the various bugs detected by three different bug-detecting tools, along with similarities in the bugs identified by each tool. As observed in the table, the issue referred to as *Measurement Related Issue*, for instance, is detected by all three tools, though each tool uses a different naming convention: QChecker identifies it as Measurement Related Issue, QSmell labels it as Intermediate Measurement, and QPAC refers to it as Incorrect Measurement. QChecker, QSmell, and QPAC are validation tools designed to address specific challenges in quantum programming within the Qiskit library. While their collective aim is to enhance the reliability of quantum programs, each employs distinct methodologies and analyses. QSmell utilizes dynamic and static analyses to detect smells unique to quantum states, whereas QChecker rapidly generates diagnostic signals for issue discovery through static analysis approaches. Conversely, QPAC identifies bug-fix patterns in Qiskit code using regular expressions and AST. QPAC focuses on enhancing speed by generating AST once for each pair of buggy-fixed code, while QSmell optimizes efficiency by employing static analysis for some smells and dynamic analysis for others. Despite their differences, these tools collaboratively work to bolster the resilience and dependability of quantum algorithms, offering tailored approaches to various aspects of program analysis and maintenance. The upcoming sections brief about these tools and their related bugs detected by them.

9.2.1 QChecker

Pioneering the realm of quantum programming in Qiskit, QChecker represents the inaugural bug detection tool explicitly designed for quantum programs. Developed by Zhao et al. [274], QChecker uses sophisticated techniques using static analysis to quickly produce diagnosed alerts that help developers identify possible issues in their codebase. Its efficacy in detecting a wide array of bug types has been rigorously verified through extensive testing. QChecker functions by incorporating two modules based on Abstract Syntax Trees (AST), which allow for the extraction of detailed program information. Additionally, the tool utilizes a wide range of bug detectors, strategically designed to effectively identify various patterns of bugs. As a result, it can detect eight kinds of bugs available in quantum programs, as mentioned in Table 9.1.

9.2.2 QSmell

QSmell given by Chen et al. [70] is a newly developed tool aimed at automatically detecting quantum-specific anomalies, or "smells," within quantum programs written in Qiskit's full-stack library. It presents an assortment of eight distinct smells designed especially for quantum programming. Both static and dynamic analytic techniques are used by QSmell, depending on the particular smell measure under study. The dynamic analysis uses an execution matrix produced during the quantum program's execution and is recommended for metrics such as LC, IM, ROC, CG, IQ, and IdQ. After that, QSmell analyzes the execution matrix to find specific smells and computes metrics like LC. On the other hand, smells like NC and LPQ, which depend more on the operations done on the quantum circuit than on the circuit itself, are subject to static analysis. For these cases, QSmell utilizes Python Abstract Syntax Trees (AST) to inspect the source code, computing metrics like LPQ by analyzing method calls within the code. Overall, QSmell offers a comprehensive approach to detecting and analyzing quantum-specific anomalies, thereby aiding in the maintenance of quantum programs.

9.2.3 QPAC

Nayak et al. [275] presented QPAC, a novel approach to automated quantum software bug fixing, introducing a research agenda focused on this endeavor. As an expansion of earlier research on bug-fix pattern identification, QPAC compares files that have been patched and those that haven't, extracting data unique to the patterns. It does this by using AST and regular expressions, or regex. The Qiskit code presently has seven bugs explicitly found in quantum programs successfully identified or rejected by the tool's de-

tectors. To improve generality and extensibility, QPAC's architecture and code have been refactored. This optimizes efficiency by producing AST just once for each pair of bug-fixed code. A limitation of earlier iterations is also addressed by QPAC, which reports all patterns found rather than just one for each bug-fix pair.

9.3 Proposed Methodology

This section presents an outline of the methodology employed in this study. It provides an overview of the dataset containing bugs and procedures for dataset pre-processing for each tool. Subsequently, it outlines the tool validation approach implemented to assess the quantum bug-detecting tools. Finally, it presents the detection performance metrics utilized in the evaluation. The methodology used in this study is shown in Figure 9.1.

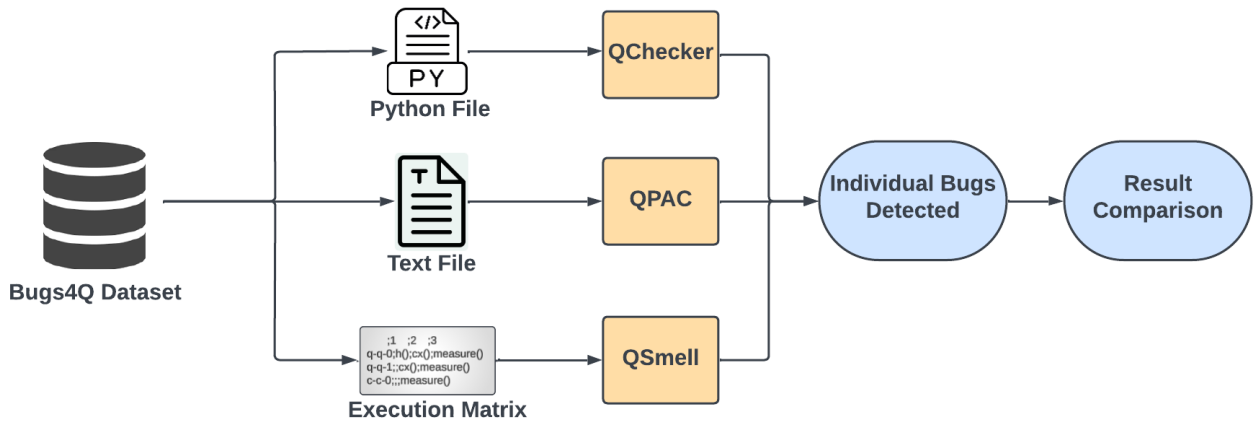


Figure 9.1: Overview of the Proposed Methodology

9.3.1 Data Pre-processing

The Bugs4Q dataset, publicly available on GitHub , originally comprises thirty-six quantum programs implemented in Qiskit, Q#, and Cirq. These programs undergo regular updates, including modifications introduced by developers to introduce bugs into the codebase. These snippets primarily represent essential segments of quantum circuits designed for various operations, ranging from basic to advanced, executed in quantum simulators.

Given that each detection tool- QChecker, QPAC, and QSmell- utilizes distinct input formats, preprocessing of the initial dataset, consisting of Python code snippets of both buggy and fixed code, needs to be conducted individually for each tool. For QChecker,

Python files containing buggy code are required. Thus, direct preprocessing of each file isn't necessary; however, ensuring that the code is executable is essential. QPAC, on the other hand, necessitates textual snippets of both buggy and fixed code. During preprocessing, the dataset should be prepared such that each file contains multiple-line textual comments containing both the buggy and fixed code segments. Finally, QSmell relies on the dynamic nature of the code. To accommodate this, each buggy code snippet requires manual insertion of an additional two-line code segment to implement the *qc2matrix* function, thereby converting the buggy code into an execution matrix stored as a CSV file. In this matrix, each row corresponds to a qubit gate operation, and each column represents the timing of the operation. This execution matrix dataset serves as the input to the QSmell tool.

By tailoring the pre-processing steps to the requirements of each detection tool, the Bugs4Q dataset can effectively support the evaluation and improvement of quantum code analysis tools.

9.3.2 Evaluation Measures Selected

This section will provide a concise overview of the comparative stages for the three tools, outlining the parameters on which they are being evaluated.

9.3.2.1 Tool Bug Detection Assessment

Firstly, all three tools will be executed on each of the 36 pre-processed files of the Bugs4Q dataset. The observed list of code smells generated by each tool will be compared with each other to assess their consistency and coverage.

9.3.2.2 Tool Reliability Assessment

As each tool may generate a different list of code smells for each file, it's essential to determine which one provides more dependable and accurate results, closely reflecting the actual code smells in the file. Therefore, the three tools were evaluated based on their true positives, false positives, true negatives, and false negatives.

9.3.2.3 Performance Metrics

Various assessment metrics have been used to evaluate the detection performance of QChecker, QSmell, and QPAC. The metrics Accuracy, Precision, Recall, and F1 Score provide information on how well each instrument predicts an outcome, as well as how well it captures important instances.

Furthermore, the MCC considers the balance between genuine and erroneous positives and negatives, making it a comprehensive assessment. These benchmarks allow researchers to compare the different homography detection tools in terms of both test performance and computational complexity, thus making it easier for researchers to decide whether one particular tool is suitable for performing a specific detection task taking into account its strengths and weaknesses.

9.3.2.4 Tool Usability Assessment

This analysis seeks to understand how approachable open-source tools are, specifically considering the kind of input they expect as well as the number of programming languages they support. Through these characteristics, developers may be able to determine how capable and experienced a tool is as well as how well it can function with other programming languages. In the end, this makes the developers able to see which of the two open-source tools have fuller features and is apt for their use case.

9.4 Results

This section evaluates how three bug detection techniques perform on a benchmark dataset. This evaluates both the way they implemented it and how well they predict bug identification. On finding strengths and weaknesses- will help decisions on future deployments.

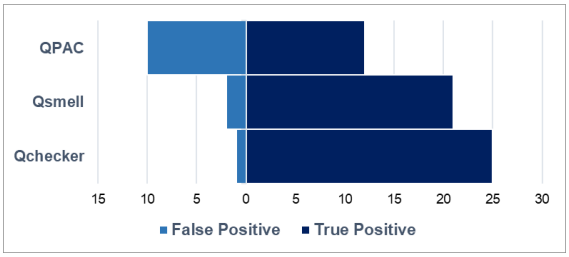
9.4.1 Tool Bug Detection Assessment

The three bug detection tools underwent testing on code snippets sourced from Bugs4Q after requisite pre-processing. Referencing Table 9.2, it's evident that QPAC struggled to identify major bugs in the majority of files. Among the 36 files tested, it only flagged bugs in three instances, while in others, it either failed to function or yielded no bug detection output. Conversely, QSmell demonstrated proficiency in analyzing both Qiskit and Q# programs, often detecting multiple bugs within them. Meanwhile, QChecker exhibited consistent performance across all files, reliably identifying bugs and occasionally reporting no issues. This table serves to elucidate the tools' detection capabilities.

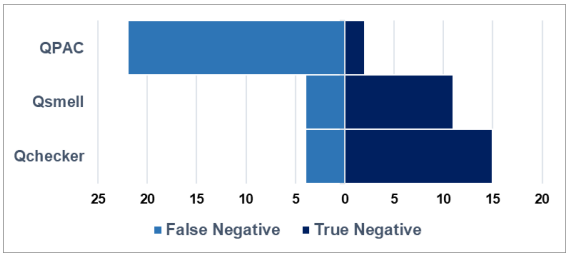
This observation underscores the superior bug detection performance of QSmell and QChecker relative to QPAC.

Table 9.2: Comparison of Quantum Bug Detection Tools on Bugs4Q Dataset

Issue No.	QSmell	QChecker	QPAC
5098	-	PE	-
356	-	PE, DO	-
537	-	-	-
596	-	PE	-
1446	-	QE	-
1985	-	-	-
3799	-	-	-
5580	-	-	-
5249	-	-	-
6892	-	IIS	-
4144	-	-	-
6571	-	-	-
135	IdQ, LC	-	-
6540	-	-	-
6255	-	-	-
664	IdQ, IQ, LC, IM	IM	Incorrect Measurement
1192	IdQ, IQ	IIS	Incorrect initialization
3954	IdQ, IQ, LC, IM	PE, IM	-
3549	IdQ, IQ, LC	CE	-
16636	-	CE	-
15711	IdQ, LC	PE	-
14945	-	IIS, PE	-
14286	IdQ, IQ, LC	PE	-
12838	IdQ, IQ	IIS	Incorrect initialization
9224	-	IIS	-
9246	-	PE	-
9871	IdQ, IQ, LC	IIS	-
4260	ROC, LC, IM	IIS	-
5557	-	CE	-
5959	-	CE	-
6692	-	-	-
6697	-	PE	-
7129	-	PE	-
8893	ROC, IdQ, IQ, LC	PE	-
9209	IdQ, IQ, LC	-	-
20894	-	DO	-



(a) True and False Positives



(b) True and False Negatives

Figure 9.2: Comparison of True and False Positives/Negatives in Detection Tools

9.4.2 Tool Reliability Assessment

By correlating the provided outputs with the actual presence of bugs in the codes, both positives and negatives are computed for further analysis as shown in Figure 9.2a and 9.2b. The analysis reveals that QChecker consistently yields the highest number of true positives and true negatives among the three tools. This indicates that QChecker's results are more accurate and closely aligned with the actual code smells present in the dataset compared to the other tools.

9.4.3 Performance Metrics

Evaluation metrics were applied to each tool, as detailed in Table 9.3. Notably, QChecker and QSmell performed comparably across all evaluation metrics, achieving accuracies of 96.4% and 95.6%, respectively. In contrast, QPAC lagged with an accuracy of 73.24%. Additionally, QPAC yielded a negative MCC value, unlike the positive values observed for QSmell and QChecker. The negative MCC value indicates that the predictions made by QPAC are poorly aligned with the actual class labels, suggesting a lack of agreement between the predicted and observed values.

Table 9.3: Performance Metrics for Each Tool

Tool	F1 Score	Recall	Precision	Accuracy	MCC
QChecker	0.9130	0.8400	1.0000	0.9640	0.5555
QSmell	0.8910	0.8005	1.0000	0.9559	0.5555
QPAC	0.6050	0.7000	0.7619	0.7324	-0.3944

9.4.4 Tool Usability Assessment

The usability of bug detection tools significantly influences their efficiency alongside performance. All three tools were accessible via open-source GitHub links offered as Python-

based software, ensuring equal accessibility and availability. Examining their input requirements, QChecker simply needed the path to Python files for analysis. In contrast, QPAC required manual entry of code snippets along with fixed code inserted as comments. QSmell had the most complexity and prerequisites, demanding Python file paths for static analysis and an execution matrix for dynamic analysis. Moreover, implementing the execution matrix code necessitated manual insertion of qc2matrix code into each Python file. Consequently, QSmell posed the most significant difficulty and prerequisites, while QChecker was the simplest to use.

Lastly, we evaluated the compatibility of the tools with quantum languages and frameworks. Table 9.4 describes the result of this analysis. The Bugs4Q dataset comprised code implemented in Qiskit, Q#, and Cirq frameworks. QChecker demonstrated versatility by successfully analyzing codes from all three frameworks. In contrast, QSmell was limited to functioning with Qiskit and Q# codes, while QPAC could only analyze Qiskit codes.

Considering the comprehensive performance analysis, it can be concluded that QChecker stands out as an effectively used tool, exhibiting both high performance and ease of use.

Table 9.4: Performance of Detecting tools on various quantum languages

Bug Detecting tool	**Availability**	**Input Required**	**Quantum Framework**		
			Qiskit	Q#	Cirq
QChecker	✓	.py file	✓	✓	✓
QPAC	✓	.txt file	✓		
QSmell	✓	.csv file	✓	✓	

9.5 Discussion

This comparative study evaluates three distinct quantum bug detection tools - QSmell, QChecker, and QPAC - using the Bugs4Q benchmark dataset. Each tool was applied to all thirty-six code snippets, and the results revealed QChecker's superior performance. Not only did QChecker exhibit the highest bug detection rate, but it also offered ease of usage and compatibility with a broader range of programming languages.

The study effectively scrutinizes the three tools, ultimately confirming QChecker as the optimal choice. For future investigations, the study proposes analyzing the tools on a larger dataset to further validate their effectiveness and robustness in real-world scenarios.

Chapter 10

Conclusion, Future Work and Social Impact

10.1 Summary of Work

This thesis presents a comprehensive investigation into code smell detection and software quality assessment across both classical and quantum software systems. The research addresses several long-standing challenges in software maintenance, including class imbalance, feature redundancy, limited utilization of structural-semantic relationships, and the absence of mature quality assessment frameworks for quantum programs. Through a combination of machine learning, deep learning, graph neural networks, quantum-inspired optimization, and empirical software engineering techniques, the thesis develops a suite of methodologies that collectively advance the state of the art in automated software quality analysis.

The first contribution of this thesis focuses on improving classical code smell detection through ensemble-based deep learning frameworks. Existing studies predominantly rely on either handcrafted software metrics or individual deep learning architectures, both of which suffer from limitations in capturing diverse code characteristics. To address this challenge, an ensemble framework integrating CNN, LSTM, and GRU architectures was proposed. The rationale behind the ensemble design was to leverage the complementary strengths of convolutional networks in feature extraction and recurrent networks in capturing sequential dependencies. Extensive experimentation demonstrated that the ensemble models consistently outperformed individual architectures across multiple code smell categories. The proposed framework improved classification performance while simultaneously enhancing robustness against dataset variability. The findings established that ensemble deep learning can serve as an effective mechanism for extracting rich soft-

ware representations and improving antipattern prediction accuracy.

While traditional machine learning and deep learning approaches primarily rely on software metrics or sequential code representations, they often fail to capture structural and semantic relationships present in software systems. To overcome this limitation, this thesis proposed *SmellGraph*, a hybrid graph neural network framework that models software systems as interconnected graphs. The framework combines structural dependencies and semantic information to create enriched graph representations of source code. Through graph-based message passing and representation learning, SmellGraph effectively captures contextual relationships among software entities. Experimental evaluation demonstrated superior performance on multi-label code smell detection tasks, highlighting the importance of integrating structural and semantic information into software quality assessment frameworks.

Class imbalance remains one of the most persistent challenges in software defect prediction and code smell detection. Many software quality datasets contain significantly fewer defective instances than non-defective instances, leading to biased learning models. To address this issue, this thesis introduced *DIR-SMOTE*, a novel density-influenced resampling strategy designed specifically for software engineering datasets. DIR-SMOTE generates synthetic instances while preserving the local distributional characteristics of minority classes. Experimental results demonstrated that the proposed approach significantly improved classification performance across both machine learning and deep learning models. The study established that effective imbalance handling is essential for building reliable and generalizable software quality prediction systems.

As quantum computing continues to mature, software quality assessment techniques must evolve to accommodate quantum programming paradigms. Existing code smell detection frameworks are primarily designed for classical systems and cannot adequately capture the unique characteristics of quantum programs. To address this gap, this thesis proposed *QUAMET*, the first metric-based framework specifically designed for quantum code smell detection. QUAMET introduces structural and semantic quality indicators tailored to quantum programs and validates their effectiveness using the LintQ dataset. The framework provides a systematic mechanism for identifying quality issues in quantum software while establishing a foundation for future research in quantum software maintainability. This contribution represents one of the earliest efforts toward formalizing software quality assessment in quantum programming environments.

Building upon the foundations established by QUAMET, this thesis proposed *QNet*, a deep learning-based hybrid ensemble framework for quantum code smell detection. Similar to the classical ensemble framework, QNet combines CNN, LSTM, and GRU architectures to capture diverse characteristics of quantum programs. The framework was

evaluated against several baseline approaches, including transfer-learning models such as BERT. Experimental results demonstrated that QNet consistently achieved superior performance across multiple evaluation metrics. The findings confirmed that deep learning techniques can be successfully adapted to quantum software quality assessment and can provide substantial improvements over conventional approaches.

Feature redundancy and high-dimensional representations often reduce the effectiveness of machine learning models. To address this challenge, this thesis proposed a modified Quantum-Inspired Evolutionary Algorithm (*m-QIEA*) for feature selection. The proposed approach incorporates enhanced exploration and exploitation mechanisms to identify compact yet informative feature subsets. Evaluation on both classical (MLCQ) and quantum datasets demonstrated that m-QIEA effectively reduced feature redundancy while improving classification accuracy and model generalization. The results highlighted the potential of quantum-inspired optimization techniques for solving complex software engineering problems involving high-dimensional data.

In addition to code smell detection, this thesis investigated bug detection in quantum software systems through an empirical evaluation of three state-of-the-art tools: QChecker, QSmell, and QPAC. Using the Bugs4Q benchmark dataset, the study assessed the capabilities of these tools in identifying defects across diverse quantum programs. The comparative analysis revealed the strengths and limitations of each tool, highlighting areas where current quantum debugging solutions remain inadequate. The findings provide valuable insights for both researchers and practitioners while establishing a benchmark for future development of automated quantum software quality assurance tools.

The major contributions of this thesis can be summarized as follows:

- Proposed ensemble-based CNN-LSTM-GRU frameworks for classical code smell detection.
- Developed SmellGraph, a hybrid graph neural network framework integrating structural and semantic code representations.
- Introduced DIR-SMOTE, a novel density-influenced resampling strategy for handling class imbalance.
- Proposed QUAMET, the first metric-based framework for quantum code smell detection.
- Developed QNet, a deep learning ensemble model for quantum code smell prediction.
- Proposed m-QIEA, a modified quantum-inspired evolutionary algorithm for feature selection.

- Conducted a comprehensive empirical evaluation of QChecker, QSmell, and QPAC using the Bugs4Q dataset.

- Established a unified software quality assessment perspective spanning both classical and quantum software systems.

Collectively, these contributions establish a robust, scalable, and interpretable foundation for automated antipattern detection in both classical and quantum software systems. The proposed datasets, algorithms, frameworks, and toolchains provide valuable resources for researchers and practitioners while advancing the state-of-the-art in software maintainability assessment.

Furthermore, the research addresses data-level, model-level, and tool-level challenges in software quality analysis. The proposed approaches consistently improved detection accuracy, robustness, and interpretability while extending software quality assessment into the emerging domain of quantum computing. Overall, this work advances the methodological foundations of software maintenance and opens new directions at the intersection of software engineering, machine learning, and quantum computing.

10.2 Applications of the Proposed Work

10.2.1 Industrial Applications

The outcomes of this research have significant implications for industrial software development, where software quality directly influences reliability, maintainability, security, and operational costs.

In classical software systems, the proposed code smell detection frameworks can assist organizations in reducing technical debt, improving maintainability, and preventing costly system failures. By identifying design flaws and implementation issues at an early stage, these approaches enable the development of cleaner and more robust software systems that can evolve efficiently with changing requirements. Consequently, organizations can achieve reduced maintenance costs, improved productivity, and enhanced software reliability.

The proposed techniques are particularly valuable in safety-critical and mission-critical environments such as healthcare, transportation, aerospace, finance, and public infrastructure, where software correctness is essential. Automated smell and bug detection can help identify hidden defects and poor design practices before deployment, thereby supporting regulatory compliance, operational safety, and system dependability.

The industrial significance becomes even more pronounced in quantum computing environments. Current quantum hardware, particularly Noisy Intermediate-Scale Quantum (NISQ) systems, is highly resource-constrained and error-prone. Even minor inefficiencies in quantum programs may significantly degrade execution fidelity and waste scarce computational resources. Automated bug and smell detection frameworks such as QUAMET, QNet, and the evaluated bug detection tools can identify issues including redundant gates, excessive circuit depth, and improper quantum operations, thereby improving computational efficiency and resource utilization.

Potential industrial applications include:

- **Healthcare:** Quantum-enhanced drug discovery, protein folding simulations, and personalized medicine applications require highly reliable quantum software. Early identification of bugs and quality issues can accelerate scientific discoveries and improve healthcare outcomes.
- **Finance:** Quantum algorithms for portfolio optimization, risk analysis, and financial forecasting benefit from improved correctness and reliability, enabling more accurate decision-making and reducing operational risks.
- **Supply Chain and Logistics:** Optimized quantum software can support routing optimization, scheduling, and resource allocation while reducing operational costs and environmental impact.
- **Cryptography and National Security:** Reliable quantum software is essential for secure communication, post-quantum cryptography, and cyber-defense applications, strengthening digital sovereignty and security.

By enabling early defect detection and improving maintainability, the proposed frameworks contribute to building trustworthy, scalable, and sustainable software systems across both classical and quantum domains.

10.2.2 Research Applications

Beyond industrial deployment, the contributions of this thesis provide a strong foundation for future research in software quality assessment, machine learning, and quantum software engineering.

The proposed datasets, including those used for classical and quantum code smell detection, can serve as benchmark resources for evaluating future detection models. Similarly, the frameworks developed in this work, including DIR-SMOTE, SmellGraph, QUAMET, QNet, and m-QIEA, can be extended and adapted for related software engineering tasks

such as vulnerability prediction, defect prediction, automated refactoring, software maintenance, and performance optimization.

The integration of deep learning, graph neural networks, evolutionary optimization, and quantum-inspired computation demonstrates the potential of interdisciplinary approaches for addressing complex software engineering problems. Future researchers may utilize these methodologies to investigate additional software quality attributes including maintainability, reliability, security, performance optimization, and technical debt prediction.

Furthermore, the empirical evaluation of quantum bug detection tools establishes an initial benchmark for quantum software quality assessment and provides a foundation for future investigations into quantum program analysis, testing, verification, and automated debugging.

The research also contributes to the emerging field of explainable software analytics by highlighting the importance of interpretable and developer-centric quality assessment frameworks. These contributions support the development of intelligent software engineering tools capable of providing actionable insights to developers and software architects.

10.3 Limitations of the Work

Although the proposed methodologies demonstrated significant improvements in code smell detection and software quality assessment across both classical and quantum software systems, certain limitations remain.

Firstly, the effectiveness of the developed models depends on the availability and quality of benchmark datasets. While datasets such as MLCQ, LintQ, and Bugs4Q were utilized, the scarcity of large-scale and industrial-grade datasets, particularly in the quantum domain, may limit the generalizability of the results. The proposed ensemble deep learning models, graph neural network framework, and optimization techniques require considerable computational resources for training and tuning. This may restrict their immediate deployment in resource-constrained environments or real-time development pipelines.

Furthermore, the majority of the proposed approaches rely on static code analysis and do not explicitly incorporate dynamic information such as execution traces, software evolution history, or developer activities. Consequently, certain runtime-related quality issues may not be fully captured. Although the proposed models achieved strong predictive performance, their interpretability remains limited due to the inherent complexity of deep learning architectures. Providing transparent and developer-friendly explanations

for predictions remains an open challenge.

Another limitation is the scope of the study, which primarily focuses on code smells, antipatterns, and quantum bug detection. Other software quality aspects, including security vulnerabilities, performance optimization, reliability assessment, and technical debt estimation, were beyond the scope of this research. Finally, the quantum-oriented frameworks were evaluated primarily using available repositories and benchmark datasets. The impact of hardware-specific constraints associated with current NISQ-era quantum computers, such as noise, decoherence, and qubit connectivity limitations, was not comprehensively investigated.

Despite these limitations, the proposed frameworks, algorithms, and empirical findings provide a strong foundation for future research in both classical and quantum software quality assessment.

10.4 Future Work

Although the proposed methodologies achieved promising results across both classical and quantum software systems, several opportunities remain for further investigation:

- Future research should focus on developing lightweight, scalable, and computationally efficient models suitable for deployment within real-time software development environments and continuous integration/continuous deployment (CI/CD) pipelines.

- The scope of quality assessment can be extended beyond code smells and quantum programming bugs to include software vulnerabilities, performance bottlenecks, maintainability indicators, reliability assessment, and technical debt analysis, thereby enabling a more comprehensive software quality framework.

- The availability of larger, more diverse, and realistic benchmark datasets remains a critical challenge. Future efforts should concentrate on constructing and curating datasets across both classical and quantum domains to improve model generalizability and facilitate broader validation.

- Advanced interpretability and explainability techniques should be explored to ensure that automated quality assessment systems remain transparent, trustworthy, and capable of providing actionable recommendations to developers.

- Emerging paradigms such as transfer learning, graph-based learning, large language models, and quantum-inspired optimization offer significant opportunities

for improving software quality analysis. Future studies may investigate hybrid approaches that combine these techniques to enhance detection accuracy and robustness.

- The SmellGraph framework can be extended toward heterogeneous graph representations that incorporate additional software artifacts such as version histories, developer activities, software documentation, test cases, and issue-tracking information.
- Future work may investigate adaptive and dynamic resampling techniques that respond to changing data distributions, extending the capabilities of DIR-SMOTE through hybrid augmentation and cost-sensitive learning strategies.
- The m-QIEA framework can be enhanced through integration with reinforcement learning, swarm intelligence, or advanced optimization strategies to improve exploration–exploitation balance and scalability on industrial-scale repositories.
- Future research should focus on developing larger and more diverse datasets for quantum smell and bug detection, expanding the taxonomy of quantum antipatterns, and incorporating hardware-aware fault models into quality assessment frameworks.
- The evaluated quantum bug detection tools may be integrated into intelligent development environments and automated static analysis pipelines to provide real-time feedback and early defect detection during quantum software development.
- Domain-specific transfer learning approaches for quantum programming languages remain largely unexplored. Future studies may investigate large-scale pre-training strategies based on quantum circuits and quantum software repositories, followed by fine-tuning for smell detection and bug prediction tasks.

By pursuing these directions, future research can further strengthen the foundations of classical and quantum software engineering while enabling the development of trustworthy, scalable, explainable, and socially impactful software quality assessment systems.

Appendix

Table A.1: List of Primary Studies Selected for SLR

ID	Title	Ref
S1	Fusion of deep convolutional and LSTM recurrent neural networks for automated detection of code smells	[43]
S2	Deep Multimodal Architecture for Detection of Long Parameter List and Switch Statements using DistilBERT	[166]
S3	DACOS-A Manually Annotated Dataset of Code Smells	[152]
S4	Improving the Quality of Open Source Software	[165]
S5	Web Service Anti-patterns Detection Using CNN with Varying Sequence Padding Size	[174]
S6	Using Word Embedding and Convolution Neural Network for Bug Triaging by Considering Design Flaws	[175]
S7	Code Smell Detection Using Ensemble Machine Learning Algorithms	[50]
S8	Application of Deep Learning models for Code Smell Prediction	[153]
S9	DeleSmell: Code smell detection based on deep learning and latent semantic analysis	[46]
S10	Feature Envy Detection with Deep Learning and Snapshot Ensemble	[155]
S11	How to improve deep learning for software analytics: (a case study with code smell detection)	[160]
S12	Multi-Label Code Smell Detection with Hybrid Model based on Deep Learning	[45]
S13	Detecting Code Smells with AI: a Prototype Study	[176]
S14	The automation of the detection of large class bad smell by using genetic algorithm and deep learning	[161]
S15	A machine and deep learning analysis among SonarQube rules, product, and process metrics for fault prediction	[162]
S16	Multi-granularity code smell detection using deep learning method based on abstract syntax tree	[164]
S17	An Empirical Study on Predictability of Software Code Smell Using Deep Learning Models	[149]

Continued on next page

Table A.1 – continued from previous page

ID	Title	Ref
S18	Two-Pass Technique for Clone Detection and Type Classification Using Tree-Based Convolution Neural Network	[163]
S19	FCCA: Hybrid Code Representation for Functional Clone Detection Using Attention Networks	[177]
S20	MARS: Detecting brain class/method code smell based on metric attention mechanism and residual network	[49]
S21	Code smell detection by deep direct-learning and transfer-learning	[178]
S22	Deep Learning Based Code Smell Detection	[157]
S23	Feature Envy Detection based on Bi-LSTM with Self-Attention Mechanism	[156]
S24	Deep Representation Learning for Code Smells Detection using Variational Auto-Encoder	[276]
S25	Deep Learning Anti-Patterns from Code Metrics History	[159]
S26	Deep semantic-Based Feature Envy Identification	[141]
S27	Detecting code smells using deep learning	[150]
S28	On the feasibility of transfer-learning code smells using deep learning	[151]
S29	Deep learning based feature envy detection	[154]
S30	A hybrid approach to detect code smells using deep learning	[158]
S31	Cclearner: A deep learning-based clone detection approach	[277]
S32	Finding bad code smells with neural network models	[180]
S33	Deep learning code fragments for code clone detection	[278]
S34	Some code smells have a significant but small effect on faults	[279]
S35	What you like in design use to correct bad-smell	[25]

Table A.2: Detailed Description of Selected Studies for the Review

ID	Author	Affiliation	Year	Publishe	Publication Type	Ref
S1	Ho et al.	Academia	2023	ACM	Conference	[43]
S2	Bhave and Sinha	Academia	2023	IEEE	Conference	[166]
S3	Nandini et al.	Industry	2023	ArXiv	Journal	[152]
S4	Kaur and Singh	Academia	2023	Wiley	Journal	[165]
S5	Tummalapalli et al.	Academia	2023	Springer	Journal	[174]
S6	Sepahvand et al.	Academia	2023	Elsevier	Journal	[175]
S7	Dewangan et al.	Academia	2022	MDPI	Journal	[50]
S8	Tarwani and Chug	Academia	2022	IEEE	Conference	[153]
S9	Zhang et al.	Academia	2022	Elsevier	Journal	[46]
S10	Zhang et al.	Academia	2022	IEEE	Conference	[155]
S11	Yedida and Menzies	Both	2022	IEEE	Conference	[160]
S12	Li and Zhang	Academia	2022	IEEE	Conference	[45]
S13	Virmajoki et al.	Academia	2022	IEEE	Conference	[176]
S14	Imam et al.	Academia	2022	Elsevier	Journal	[161]
S15	Lomio et al.	Both	2022	Springer	Journal	[162]
S16	Xu and Zhang	Academia	2021	SEKE	Conference	[164]
S17	Gupta et al.	Academia	2021	Springer	Conference	[149]
S18	Jo et al.	Academia	2021	MDPI	Journal	[163]
S19	Wei et al.	Academia	2021	IEEE	Journal	[177]
S20	Zhang and Dong	Academia	2021	Wiley	Journal	[49]
S21	Sharma et al.	Both	2021	Elsevier	Journal	[178]
S22	Liu et al.	Academia	2021	IEEE	Journal	[157]
S23	Wang et al.	Academia	2020	IEEE	Conference	[156]
S24	Hadj and Bouassida	Academia	2019	IEEE	Conference	[276]
S25	Barbez et al.	Industry	2019	IEEE	Conference	[159]
S26	Guo et al.	Academia	2019	ACM	Conference	[141]
S27	Das et al.	Industry	2019	IEEE	Conference	[150]
S28	Sharma et al.	Both	2019	ArXiv	Journal	[151]

Continued on next page

Table A.2 – continued from previous page

ID	Author	Affiliation	Year	Publishe	Publication Type	Ref
S29	Liu et al.	Academia	2018	IEEE	Conference	[154]
S30	Hadj and Bouassida	Academia	2018	ENASE	Proceedings	[158]
S31	Li et al.	Academia	2017	IEEE	Conference	[277]
S32	Kim	Academia	2017	IAES	Journal	[180]
S33	White et al.	Academia	2016	IEEE	Proceedings	[278]
S34	Hall et al.	Academia	2014	ACM	Journal	[279]
S35	Kessentini et al.	Academia	2013	Springer	Journal	[25]

Table B.1: Project-Wise Code Smell values from QNet

Projects	QSmell				
	LC	IdQ	IQ	IM	ROC
Bell State Circuit	6	0	0	0	0
Bernstein-Vazirani Circuit	20	3	3	0	0
Circuit Draw	6	0	0	0	0
Commutation Relation	4	0	0	0	0
Five-Qubit GHZ Circuit	30	3	3	0	0
GHZ	4	0	0	0	0
Initialize	4	0	0	0	0
Grover	42	2	1	0	1
Load Qasm	24	0	0	0	0
Rippleadd	40	6	3	0	0
Stochastic Swap	20	1	0	1	0
Teleport	20	1	0	1	0
Three-Qubit GHZ	6	0	0	0	0
Using Qiskit Terra Level	6	0	0	0	0
Quantum Circuit.draw	2	0	0	0	0
Deutsch-Jozsa Circuit	12	1	1	0	0
Quantum Fourier Transform	18	2	2	0	0
Superdense Coding	10	1	0	0	0
Swap Test	6	0	0	0	0
MLAE	1	0	0	0	0
Adapt VQE	18	7	6	0	0
AE	4	0	0	1	0
FAE	4	1	1	0	0
IAE	3	0	0	0	0
IPE	24	0	0	0	3
QAOA	18	1	1	0	0
QEOM	30	3	0	0	0
Multi-Controlled CNOT	5	0	0	0	0
Composite Gates	6	0	0	0	0
Parameterized Circuits	30	3	0	0	0
Reducing Compilation Cost	360	1	1	0	6
Composition	8	0	0	0	0

Continued on next page

Table B.1 – continued from previous page

Projects	QSmell				
	LC	**IdQ**	**IQ**	**IM**	**ROC**
Mitigation	3	0	0	0	0
Mitigation1	4	0	0	0	0
Mitigation2	4	0	0	0	0
Mitigation3	11	0	0	0	0
HHL	2	0	0	0	0
QGAN	4	0	0	0	0
QSVC	4	0	0	0	0
Shor	19852	15650	15650	0	0
$T\Phi$ Characterization	3	0	0	0	0
$T\Phi$ Characterization1	6	0	0	0	1
$T\Phi$ Characterization2	5	0	0	0	0
T2 Hahn Characterization	6	0	0	0	1
T2 Hahn Characterization1	4	0	0	0	0
T2 Hahn Characterization2	6	0	0	0	1
Randomized Benchmarking	4	0	0	0	0
Randomized Benchmarking1	7	0	0	0	0
Randomized Benchmarking2	9	0	0	0	1
T2 Ramsey Characterization	5	0	0	0	0

List of Publications and their Proofs

Journal Publications

Journal Paper 1: Examining deep learning's capability to spot code smells: a systematic literature review

Cluster Computing (2023) 26:3473–3501
https://doi.org/10.1007/s10586-023-04144-1

Examining deep learning's capability to spot code smells: a systematic literature review

Ruchika Malhotra[1] · Bhawna Jain[1,2] · Marouane Kessentini[3]

Received: 20 July 2023 / Revised: 26 August 2023 / Accepted: 1 September 2023 / Published online: 3 October 2023
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

Abstract
Code smells violate software development principles that make the software more prone to errors and changes. Researchers have developed code smell detectors using manual and semi-automatic methods to identify these issues. However, three key challenges have limited the practical use of these detectors: developers' subjective perceptions of code smells, lack of consensus in the detection process, and difficulty in setting appropriate detection thresholds. While code smell detection using machine learning has progressed significantly, there still appears to be a gap in understanding the effective utilization of deep learning (DL) approaches. This paper aims to review and identify current methods for code smell detection using DL techniques. A systematic literature review is conducted on 35 primary studies from a collection of 8739 publications between 2013 and the present. The analysis reveals that common code smells detected include Feature Envy, God Classes, Long Methods, Complex Classes, and Large Classes. The most popular DL algorithms used are Recurrent Neural Networks (RNN) and Convolutional Neural Networks (CNN), often combined with other techniques for better results. Algorithms that train models on large datasets with fewer independent variables demonstrate exemplary performance. The paper also highlights open issues and provides guidelines for future metric identification and selection research.

Keywords Antipatterns · Code smells · Deep learning · Software smells · Smell detection

1 Introduction

With the advent of technological advancements in the industry, software maintenance is becoming a crucial phase of the software development life cycle. Software systems are changed to fix bugs, enhance base features or add new requirements [1]. But due to time constraints and early deliverables, these changes are frequently performed carelessly, resulting in the introduction of "technical debt" in the code and degrading the system's original design. Technical debt refers to the presence of "antipatterns or smells" in the software. Generally, smells causes problems in two sections of development- either at the code part or the design part. The former refers to a problem with code writing and formulation [2], commonly known as "code smells", while the latter refers to issues with the designing of the software itself and is termed as model smells or design flaws [3]. Amongst the two, *Code Smells* is considered more severe, as they not only worsen developers' comprehension of the source code [4], but their emergence also affects the object classes making them more fault-prone [5, 6]. In 1998, Brown et al. [7] also explained the defects caused due to antipatterns in software maintenance. Table 1 defines a short description of such code smells. Although these defects don't always yield the program to act wrong or crash, they make it more challenging to comprehend and maintain [8]. As a result, they increase the expense and effort for maintainability [9, 10]. According to earlier research, the costs of the maintenance stage range from about 50% to 80% of the entire cost. If technical debts are not managed correctly, they can even lead to the failure of a software project by creating interest that eats up the money intended for new features. As a result, the research

✉ Bhawna Jain
bhawnajain@igdtuw.ac.in

[1] Delhi Technological University, Delhi, India

[2] Indira Gandhi Delhi Technical University for Women, Delhi, India

[3] University of Michigan-Flint College of Innovation & Technology, Michigan, USA

Journal Paper 2: QNet: exploring deep learning for quantum code smell detection

Software Quality Journal (2025) 33:25
https://doi.org/10.1007/s11219-025-09724-5

RESEARCH

QNet: exploring deep learning for quantum code smell detection

Ruchika Malhotra[1] · **Bhawna Jain**[1,2] · **Marouane Kessentini**[3]

Accepted: 19 May 2025 / Published online: 27 May 2025
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2025

Abstract
Quantum computing (QC) has surged as a burgeoning domain, driving the evolution of novel programming paradigms. Despite extensive exploration, a critical aspect remains underexplored: *detecting code smells in quantum programs (QPs)*. Code smells, indicative of potential maintenance challenges, have been extensively studied in classical programming but pose unique hurdles in the quantum realm due to inherent disparities caused by unstable states in QC. This paper proposes a novel approach leveraging deep learning (DL) techniques for detecting quantum code smells (QCS). A comprehensive ablation study compares DL methodologies, including Convolutional Neural Networks (CNN), Gated Recurrent Units (GRU), and Long Short-Term Memory (LSTM), and proposes a hybrid model combining CNN, LSTM, and GRU layers. Critical research questions are posed regarding QCS influence on error rates in quantum computers, the relative impact of different QCS on quantum system performance, and the complementary behaviour of distinct DL models in QCS detection. Through manual curation of a labelled dataset comprising 136 open-source projects, quantum circuits are extracted and analyzed. When evaluated on the quantum dataset, this hybrid model outperforms single-layer models. Furthermore, a comparative analysis with a transfer learning (TL) approach employing a pre-trained Bidirectional Encoder Representation Transformers (BERT) model underscores the superiority of the proposed DL-based solution. The proposed model achieves an impressive accuracy rate of **92.86%**, surpassing existing DL and TL approaches. In conclusion, this research demonstrates the potential of DL to identify QCS, with the hybrid model offering avenues for further discoveries in QCS detection using DL.

Keywords Code smells · Deep learning · Quantum-specific code smell · Smell detection · Quantum computing

✉ Bhawna Jain
dr.bhawnajain.e@gmail.com

Ruchika Malhotra
ruchikamalhotra@dtu.ac.in

Marouane Kessentini
kessentm@gvsu.edu

1 Delhi Technological University, Delhi, India

2 Indira Gandhi Delhi Technical University for Women, Delhi, India

3 Grand Valley State University, Michigan, USA

Journal Paper 3: DIR-SMOTE: A Density-Influence Resampling Framework for Imbalanced Code Smell Detection



SPRINGER NATURE Link

Log in

Find a journal | Publish with us | Track your research | Search | Saved research | Cart

Home > Automated Software Engineering > Article

DIR-SMOTE: a density-influence resampling framework for imbalanced code smell detection

Published: 04 May 2026

Volume 33, article number 76, (2026) Cite this article

Save article

Automated Software Engineering

Aims and scope →

Submit manuscript →

Ruchika Malhotra, Bhawna Jain & Marouane Kessentini

Abstract

Code smell detection is vital for ensuring software quality, but the imbalance between smelly and non-smelly code instances impairs detection, especially for minority smells like Data Class and Feature Envy. Existing oversampling techniques, such as Synthetic Minority Oversampling Technique (SMOTE), Borderline-SMOTE (BL-SMOTE), and Adaptive Synthetic (ADASYN), attempt to mitigate this issue but often introduce noise or semantically irrelevant samples. This study proposes DIR-SMOTE (Density and Influence-based Resampling using SMOTE), a density and explanation-guided resampling framework that integrates local density estimation and SHapley Additive exPlanations (SHAP)-based feature importance to improve the quality of synthetic minority samples. Initially, DIR-SMOTE filters out noisy or isolated minority instances using density metrics. It then employs SHAP to identify the most influential features per instance. Synthetic samples are generated by interpolating between dense neighbors while perturbing only top-ranked SHAP features, thereby preserving semantic integrity. DIR-SMOTE is evaluated on five benchmark datasets, namely, Apache, jEdit, EDTForCSD, DesigniteJava, and MLCQ, across multiple smells such as Long Method, Feature Envy, and Data Class. Compared to nine standard resampling methods, DIR-SMOTE achieves up to 6.7% improvement in F1-score and 5.1% in precision, consistently enhancing smelly code detection in both binary and multiclass settings. Rather than relying on complex generative models, DIR-SMOTE focuses on explanation-guided and density-aware sample generation that remains transparent and computationally efficient. Overall, it offers a lightweight and robust solution that can be seamlessly integrated into practical quality assurance workflows, including automated smell detection tools and IDE-based analyzers.

Access this article

Log in via an institution →

Subscribe and save

Springer+ from €37.37 /Month

- Starting from 10 chapters or articles per month
- Access and download chapters and articles from more than 300k books and 2,500 journals
- Cancel anytime

View plans →

Buy Now

Buy article PDF 39,95 €

Journal Paper 4: m-QIEA: A Quantum-Inspired Evolutionary Algorithm for Optimized Feature Selection in Code Smell Detection

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

m-QIEA: A modified quantum-inspired evolutionary algorithm for feature selection in code smell detection☆

Ruchika Malhotra [a], Bhawna Jain [a,b],*, Marouane Kessentini [c]

[a] Delhi Technological University, Delhi, India
[b] Indira Gandhi Delhi Technical University for Women, Delhi, India
[c] Grand Valley State University, MI, USA

ARTICLE INFO

Keywords:
Code smells
Deep learning
Quantum-specific code smell
Smell detection
Evolutionary algorithm

ABSTRACT

Code smells, indicative of structural defects in software, signify sub-optimal design practices that impede software maintainability. Detecting and mitigating code smells is essential for improving software quality, particularly in both classical and quantum systems. However, conventional machine learning (ML) approaches for code smell detection often suffer from feature redundancy, increased computational cost, and suboptimal model performance. Feature selection (FS) plays a pivotal role in enhancing code smell detection by improving classification accuracy while reducing model complexity. This research aims to improve code smell detection by introducing an FS-driven approach that optimizes feature selection to enhance classification accuracy. The proposed approach leverages deep learning (DL) models for feature extraction, followed by a modified Quantum-Inspired Evolutionary Algorithm (m-QIEA) to refine the feature subset for classification. Compared with standard QIEA, m-QIEA incorporates a time-decaying rotation schedule and Gaussian mutation, which are intended to mitigate premature stagnation during the feature-selection search. Experimental evaluations are conducted on a comprehensive suite of six datasets. For classical systems, the primary MLCQ benchmark is used and further the work is extended to SmellyCode++, and a multi-language corpus (Python, Java, and C++). For quantum systems, the primary QNet benchmark is used, extended to a derived large-scale LintQ corpus, and tested on the independent Smelly Eight benchmark. Across all settings, the results demonstrate that m-QIEA achieves superior accuracy and generalization. The study establishes the critical role of FS in optimizing code smell detection accuracy. By integrating DL for feature extraction and m-QIEA for FS, the proposed approach enhances the classification of code smells across multiple programming paradigms and dataset scales.

1. Introduction

Software maintenance is an essential aspect of software development that contributes to the longevity and durability of software systems (Santos et al., 2023; Singh and Malhotra, 2012). However, the emergence of code smells, often attributed to developers' lack of coding experience, poses a significant challenge during maintenance (Bafatakis et al., 2019; Omari and Martinez, 2020; Chen et al., 2016). The introduction of code smells in software maintenance threatens the overall durability of the software, making it difficult to interpret and modify (Sandouka and Aljamaan, 2023). In classical computing, researchers actively investigate prominent code smells like feature envy (Liu et al., 2018), god class (Alkharabsheh et al., 2022; Yadav et al., 2024b), and duplicate code (AlOmar et al., 2023). Consequently, code smell detection becomes a critical field, playing a pivotal role in enhancing the longevity and functionality of software systems (Zhang and Dong, 2024).

Similarly, the advancement of Quantum Computing (QC) and the emergence of quantum programming languages such as Cirq and Qiskit has significantly propelled the development of Quantum computers (Openja et al., 2022; Fortunato et al., 2022). Given the current limited access to scalable quantum hardware, most quantum software development and testing are conducted on quantum simulators. This progress has also brought about the emergence of code smells in Quantum programs (Yadav et al., 2024c,a; Dewangan and Rao, 2022). These code smells often manifest during simulation and can lead to performance degradation, such as increased gate count, circuit depth, and memory overhead, which directly affects the efficiency and scalability

☆ Editor: Hongyu Zhang.

* Corresponding author at: Indira Gandhi Delhi Technical University for Women, Delhi, India.
E-mail address: dr.bhawnajain.e@gmail.com (B. Jain).

https://doi.org/10.1016/j.jss.2026.113115
Received 9 April 2026; Received in revised form 14 August 2026; Accepted 19 September 2026
Available online 26 September 2026
0164-1212/© 2026 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

Conference Publications

Conference Paper 1: A Comparative Analysis of Quantum Programming Bug Detection Tools

25/08/2025, 18:33 A Comparative Analysis of Quantum Programming Bug Detection Tools | SpringerLink

SPRINGER NATURE Link

Log in

Menu Search Cart

Home > Data Science and Applications > Conference paper

A Comparative Analysis of Quantum Programming Bug Detection Tools

| Conference paper | First Online: 20 May 2025

| pp 115–127 | Cite this conference paper



Data Science and Applications

(ICDSA 2024)

Ruchika Malhotra, Bhawna Jain & Marouane Kessentini

Part of the book series: Lecture Notes in Networks and Systems ((LNNS,volume 1237))

Included in the following conference series:
International Conference on Data Science and Applications

111 Accesses

Abstract

Quantum computing represents a significant advancement in the tech industry, offering researchers a new avenue to explore through the development of novel programming languages and frameworks tailored for the quantum domain. However, as quantum programming evolves, so does the challenge of software errors and bugs introduced

https://link.springer.com/chapter/10.1007/978-981-96-1185-0_10 1/9

Conference Paper 2: An Empirical Study on Feature Extraction Using Ensemble Deep Learning for Code Smell Detection

SPRINGER NATURE Link

Log in

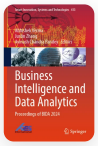
Menu Search Cart

Home > Business Intelligence and Data Analytics > Conference paper

An Empirical Study on Feature Extraction Using Ensemble Deep Learning for Code Smell Detection

| Conference paper | First Online: 25 February 2025

| pp 573–585 | Cite this conference paper



Business Intelligence and Data Analytics

(BIDA 2024)

Ruchika Malhotra, Bhawna Jain & Marouane Kessentini

Part of the book series: Smart Innovation, Systems and Technologies ((SIST,volume 413))

Included in the following conference series: International Conference on Business Intelligence and Data Analytics

270 Accesses

Abstract

Code smell detection involves identifying poorly designed and implemented code segments, which has become a focal point of research. Recently, Deep Learning (DL)

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi-42

PLAGIARISM VERIFICATION

Title of the Thesis: AN EMPIRICAL STUDY ON ANTIPATTERNS PREDICTABILITY USING MACHINE LEARNING TECHNIQUES FOR ENHANCING SOFTWARE MAINTENANCE

Total Pages: 358 **Name of the Scholar:** BHAWNA JAIN

Supervisor(s)

(1) **Prof. Ruchika Malhotra**

(2) **Dr. Marouane Kessentini**

Department: SOFTWARE ENGINEERING

Discipline: COMPUTER SCIENCE

This is to report that the above thesis was scanned for similarity detection. Process and outcome is given below:

Software used: TURNITIN

Similarity Index: 7% **Total Word Count:** 1,09,968 words

Date: October 11, 2026

Candidate's Signature

kessentini

Signature of Supervisor(s)

Plagiarism Report

7% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

Filtered from the Report

- Bibliography
- Quoted Text

Exclusions

- 3 Excluded Sources

Match Groups

668 Not Cited or Quoted 6%
Matches with neither in-text citation nor quotation marks

74 Missing Quotations 1%
Matches that are still very similar to source material

0 Missing Citation 0%
Matches that have quotation marks, but no in-text citation

0 Cited and Quoted 0%
Matches with in-text citation present, but no quotation marks

Top Sources

3% Internet sources
6% Publications
1% Submitted works (Student Papers)

Integrity Flags

1 Integrity Flag for Review

Replaced Characters
116 suspect characters on 45 pages
Letters are swapped with similar characters from another alphabet.

Our system's algorithms look deeply at a document for any inconsistencies that would set it apart from a normal submission. If we notice something strange, we flag it for you to review.

A Flag is not necessarily an indicator of a problem. However, we'd recommend you focus your attention there for further review.

turnitin Page 2 of 371 - Integrity Overview Submission ID trn:oid:::1:3559959576

Supervisors' Biography



Prof. Ruchika Malhotra
Professor & Head
Department of Software Engineering
Delhi Technological University

Educational Qualifications:
Postdoc (Indiana University–Purdue University Indianapolis, USA),
Ph.D. (Computer Applications), M.Tech. (Software Engineering)

Prof. Ruchika Malhotra is Professor and Head of the Department of Software Engineering at Delhi Technological University, Delhi, India. She pursued postdoctoral research at Indiana University–Purdue University Indianapolis, USA, under the prestigious Raman Fellowship. She received her master's and doctorate degrees in Software Engineering from Guru Gobind Singh Indraprastha University, Delhi, India. She has been recognized with the IBM Faculty Award (2013) and the DTU Commendable Research Award for six consecutive years (2018–2023). Prof. Malhotra is consistently listed among the world's top 2% scientists by Stanford University (USA) for her contributions in the fields of *Artificial Intelligence and Image Processing*. Her research interests include software testing, software quality improvement, software metrics, statistical and adaptive prediction models, and empirical software engineering. She has published more than 245 research papers in international journals and conferences. Her H-index, as reported by Google Scholar, is 40. She is the author of the book *Empirical Research in Software Engineering* (CRC Press) and co-author of *Object-Oriented Software Engineering* (PHI Learning).



Dr. Marouane Kessentini
Dean, College of Computing
Grand Valley State University, USA
Director, NSF IUCRC Centre on Pervasive AI (Michigan Site)

Educational Qualifications:
Ph.D. in Computer Science, University of Paris, Sorbonne, France

Dr. Marouane Kessentini is Dean of the College of Computing at Grand Valley State University and Director of the NSF IUCRC Centre on Pervasive AI (PPI, Michigan Site). Previously, he served as Myron and Margaret Winegarden Professor, Associate Dean for Research and Graduate Studies, and Professor of Computer Science at the College of Innovation and Technology, University of Michigan-Flint. He is the recipient of the 2018 President of Tunisia Distinguished Research Award, the University Distinguished Teaching Award, the University Distinguished Digital Education Award, the College of Engineering and Computer Science Distinguished Research Award, and four Best Paper awards, including an IEEE 10-Year Most Influential Paper Award (2011–2021). His AI-based software refactoring invention has been licensed and deployed by several Fortune 500 companies and was recognized as one of the Top 8 inventions at the University of Michigan in 2018 by the UM Technology Transfer Office. Dr. Kessentini has received research grants from industry and federal agencies and published more than 175 papers in top journals and conferences. His h-index, as reported by Google Scholar, is 50.

Author's Biography



Ms. Bhawna Jain
Ph.D. Scholar (Computer Science) at Delhi Technological University (DTU)
Assistant Professor, Department of CSE
Indira Gandhi Delhi Technical University for Women (IGDTUW), India

Educational Qualifications:
M.Tech. (Software Engineering), B.Tech. (Computer Science and Engineering)

Ms. Bhawna Jain is pursuing her Ph.D. in Computer Science under joint supervision from Delhi Technological University (DTU), India, and the University of Michigan–Flint, USA. She is currently an Assistant Professor in the Department of Computer Science and Engineering at the Indira Gandhi Delhi Technical University for Women (IGDTUW). She also has more than three years of professional experience as a Software Developer in the industry, during which she received several recognitions including the All Rounder Performer Award and the ACE Excellence Award for her role as a Growth Catalyst. She was also appreciated for her contributions to the R&D department. She has contributed to the Guinness World Record for the "Largest Online Video Chain of People Handing Over an Object." Her research work has been published with IEEE, Elsevier and Springer. Her areas of interest include Quantum Computing, Artificial Intelligence, Deep Learning, Software Maintenance, and Generative AI.

Bibliography

[1] L. Rokach, "Ensemble-based classifiers," *Artificial Intelligence Review*, vol. 33, no. 1-2, pp. 1–39, 2010.

[2] K. Han and J. He, "Quantum-inspired evolutionary algorithm for a class of combinatorial optimization," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 6, pp. 580–593, 2002.

[3] T. G. Dietterich, "Ensemble methods in machine learning," in *International Workshop on Multiple Classifier Systems*. Springer, 2000, pp. 1–15.

[4] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[5] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[6] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[7] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[8] T. Bäck, *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996.

[9] R. Olson, N. Bartley, R. Urbanowicz, and J. H. Moore, "Tpot: A tree-based pipeline optimization tool for automating machine learning," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2016, pp. 485–492.

[10] M. Ali, "Pycaret: An open source, low-code machine learning library in python," *Journal of Machine Learning Research*, vol. 21, pp. 1–10, 2020.

[11] A. Lederman and E. LeDell, "H2o: A platform for data science and machine learning," H2O.ai, Tech. Rep., 2015.

[12] Z. Xu and Q. Zhang, "A comprehensive review on automl: Concepts, approaches, and trends," *International Journal of Machine Learning and Applications*, vol. 13, no. 4, pp. 1103–1123, 2021.

[13] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of IEEE International Conference on Neural Networks*, 1995, pp. 1942–1948.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[15] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall, 2002.

[16] A. Koenig and R. P. Gabriel, "Antipatterns: New terminology or old concepts?" Presented at the OOPSLA '95 Conference, 1995, object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).

[17] W. J. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. New York: John Wiley and Sons, 1998.

[18] F. Arcelli Fontana, I. Pigazzini, R. Roveda, D. A. Tamburri, M. Zanoni, and E. Di Nitto, "Arcan – a tool for architectural smells detection," in *IEEE International Conference on Software Architecture (ICSA) Workshops*, 2017, pp. 282–285.

[19] A. Tommasel, J. A. Díaz-Pace, and D. Godoy, "Aspredictor: A tool for architectural smell prediction," GitHub repository, 2019, available at https://github.com/tommantonela/ASPredictor.

[20] W. S. Cunha, G. A. Armijo, and V. V. de Camargo, "Inset: A tool to identify architecture smells using machine learning," in *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, 2020, pp. 760–765.

[21] M. Fowler, *Improving the Design of Existing Code*. Addison-Wesley, 1999.

[22] M. Fowler, "Refactoring: Improving the design of existing code," in *Extreme Programming and Agile Methods — XP/Agile Universe 2002*, D. Wells and L. Williams, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 256–256.

[23] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2008, pp. 329–331.

[24] M. Mantyla, "Bad smells in software-a taxonomy and an empirical study," Ph.D. dissertation, PhD thesis, Helsinki University of Technology, 2003.

[25] M. Kessentini, R. Mahaouachi, and K. Ghedira, "What you like in design use to correct bad-smells," *Software Quality Journal*, vol. 21, pp. 551–571, 2013.

[26] H. Liu, Q. Liu, Z. Niu, and Y. Liu, "Dynamic and automatic feedback-based threshold adaptation for code smell detection," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 544–558, 2015.

[27] W. C. Wake, *Refactoring workbook*. Addison-Wesley Professional, 2004.

[28] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: research and practice*, vol. 23, no. 3, pp. 179–202, 2011.

[29] H. Alsolai and M. Roper, "A systematic literature review of machine learning techniques for software maintainability prediction," *Information and Software Technology*, vol. 119, p. 106214, 2020.

[30] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, 2016, pp. 1–12.

[31] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *Journal of Software: Evolution and Process*, vol. 27, no. 11, pp. 867–895, 2015.

[32] A. Alazba and H. Aljamaan, "Code smell detection using feature selection and stacking ensemble: An empirical investigation," *Information and Software Technology*, vol. 138, p. 106648, 2021.

[33] B. Van Oort, L. Cruz, M. Aniche, and A. Van Deursen, "The prevalence of code smells in machine learning projects," in *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*. IEEE, 2021, pp. 1–8.

[34] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 612–621.

[35] D. Mesbah, N. El Madhoun, K. Al Agha, and H. Chalouati, "Beyond the code: unraveling the applicability of graph neural networks in smell detection," in *International Conference on Network-Based Information Systems*. Springer, 2024, pp. 148–161.

[36] H. ZHANG and T. KISHI, "Large class detection using gnns: A graph based deep learning approach utilizing three typical gnn model architectures," *IEICE Transactions on Information and Systems*, vol. E107.D, no. 9, pp. 1140–1150, 2024.

[37] D. Yu, Y. Xu, L. Weng, J. Chen, X. Chen, and Q. Yang, "Detecting and refactoring feature envy based on graph neural network," in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, 2022, pp. 458–469.

[38] D. Yu, Y. Xu, L. Weng, J. Chen, X. Chen, and Q. Yang, "Efficient feature envy detection and refactoring based on graph neural network," *Automated Software Engineering*, vol. 32, no. 1, p. 7, 2025.

[39] R. D. PRASAD and M. SRIVENKATESH, "A hybrid model combining graph neural networks, reinforcement learning, and autoencoders for automated code refactoring and optimization," *Journal of Theoretical and Applied Information Technology*, vol. 103, no. 1, 2025.

[40] H. Zhang and T. Kishi, "Long method detection using graph convolutional networks," *Journal of Information Processing*, vol. 31, pp. 469–477, 2023.

[41] R. Sandouka and H. Aljamaan, "Python code smells detection using conventional machine learning models," *PeerJ Computer Science*, vol. 9, p. e1370, 2023.

[42] M. Hadj-Kacem and N. Bouassida, "Application of deep learning for code smell detection: Challenges and opportunities," *SN Computer Science*, vol. 5, no. 5, p. 614, 2024.

[43] A. Ho, A. M. T. Bui, P. T. Nguyen, and A. Di Salle, "Fusion of deep convolutional and lstm recurrent neural networks for automated detection of code smells," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 229 234. [Online]. Available: https://doi.org/10.1145/3593434.3593476

[44] W. Ma, Y. Yu, X. Ruan, and B. Cai, "Pre-trained model based feature envy detection," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023, pp. 430–440.

[45] X. Li, Y. Wang *et al.*, "Multi-label code smell detection with hybrid model based on deep learning," in *International Conference on Software Engineering and Knowledge Engineering*. KSI Research Inc., 2022, pp. 77–82.

[46] Y. Zhang, C. Ge, S. Hong, R. Tian, C. Dong, and J. Liu, "Delesmell: Code smell detection based on deep learning and latent semantic analysis," *Knowledge-Based Systems*, vol. 255, p. 109737, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095070512200879

[47] N. Khleel, "Improving accuracy of code smells detection with data balancing techniques," *1*, 02 2022.

[48] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia, "On the role of data balancing for machine learning-based code smell detection," in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, ser. MaLTeSQuE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 24. [Online]. Available: https://doi.org/10.1145/3340482.3342744

[49] Y. Zhang and C. Dong, "Mars: Detecting brain class/method code smell based on metric–attention mechanism and residual network," *Journal of Software: Evolution and Process*, p. e2403, 2021.

[50] S. Dewangan, R. S. Rao, A. Mishra, and M. Gupta, "Code smell detection using ensemble machine learning algorithms," *Applied sciences*, vol. 12, no. 20, p. 10321, 2022.

[51] S. Boutaib, S. Bechikh, F. Palomba, M. Elarbi, M. Makhlouf, and L. B. Said, "Code smell detection and identification in imbalanced environments," *Expert Systems with Applications*, vol. 166, p. 114076, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417420308356

[52] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. Chikha, "Competitive coevolutionary code-smells detection," vol. 8084, pp. 50–65, 08 2013.

[53] G. Saranya, H. Khanna Nehemiah, A. Kannan, and V. Nithya, "Model level code smell detection using egapso based on similarity measures," *Alexandria Engineering Journal*, vol. 57, no. 3, pp. 1631–1642, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S111001681730234X

[54] H. Palangi, L. Deng, Y. Shen, J. Gao, X. He, J. Chen, X. Song, and R. Ward, "Deep sentence embedding using long short-term memory networks: Analysis and application to information retrieval," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 24, no. 4, pp. 694–707, 2016.

[55] S. Guessoum, S. Belda, J. M. Ferrandiz, S. Modiri, S. Raut, S. Dhar, R. Heinkelmann, and H. Schuh, "The short-term prediction of length of day using 1d convolutional neural networks (1d cnn)," *Sensors*, vol. 22, no. 23, 2022. [Online]. Available: https://www.mdpi.com/1424-8220/22/23/9517

[56] A. Ghannem, G. El Boussaidi, and M. Kessentini, "On the use of design defect examples to detect model refactoring opportunities," *Software Quality Journal*, vol. 24, pp. 947–965, 2016.

[57] F. L. Caram, B. R. D. O. Rodrigues, A. S. Campanelli, and F. S. Parreiras, "Machine learning techniques for code smells detection: a systematic mapping study," *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, no. 02, pp. 285–316, 2019.

[58] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, 2019.

[59] M. Baratchi, C. Wang, S. Limmer, J. van Rijn, H. Hoos, T. Back, and M. Olhofer, "Automated machine learning: past, present and future," *Artificial Intelligence Review*, vol. 57, 04 2024.

[60] D. Al-Fraihat, Y. Sharrab, A.-R. Al-Ghuwairi, M. AlElaimat, and M. Alzaidi, "Detecting and resolving feature envy through automated machine learning and move method refactoring," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 14, no. 2, 2024. [Online]. Available: https://ijece.iaescore.com/index.php/IJECE/article/view/34459

[61] Z. Huang, H. Yu, G. Fan, Z. Shao, M. Li, and Y. Liang, "Aligning xai explanations with software developers expectations: A case study with code smell prioritization," *Expert Systems with Applications*, vol. 238, p. 121640, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417423021425

[62] Z. Huang, H. Yu, G. Fan, Z. Shao, Z. Zhou, and M. Li, "On the effectiveness of developer features in code smell prioritization: A replication study," *Journal of Systems and Software*, vol. 210, p. 111968, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121224000116

[63] Y. Huang and M. Martonosi, "Statistical assertions for validating patterns and finding bugs in quantum programs," in *Proceedings of the 46th International*

Symposium on Computer Architecture, 2019, p. 541 553. [Online]. Available: http://dx.doi.org/10.1145/3307650.3322213

[64] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie, "Proq: Projection-based runtime assertions for debugging on a quantum computer," in *ArXiv preprint*, 2019.

[65] J. Campos and A. Souto, "Qbugs: A collection of reproducible bugs in quantum algorithms and a supporting infrastructure to enable controlled quantum software testing and debugging experiments," 2021.

[66] M. Paltenghi and M. Pradel, "Bugs in quantum computing platforms: an empirical study," *Proceedings of the ACM on Programming Languages*, vol. 6, pp. 1 – 27, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:239998607

[67] P. Zhao, X. Wu, J. Luo, Z. Li, and J. Zhao, "An empirical study of bugs in quantum machine learning frameworks," 2023.

[68] M. Openja, M. M. Morovati, L. An, F. Khomh, and M. Abidi, "Technical debts and faults in open-source quantum software systems: An empirical study," *Journal of Systems and Software*, vol. 193, p. 111458, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222001480

[69] K. Kher, I. Joshi, B. Mukkavalli, L. Zhang, and R. Panduranga, "Automatic diagnosis of quantum software bug fix motifs," in *1*, 07 2023, pp. 97–102.

[70] Q. Chen, R. Câmara, J. Campos, A. Souto, and I. Ahmed, "The smelly eight: An empirical study on the prevalence of code smells in quantum computing," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 358–370.

[71] L. De Stefano, C. A. Furia, and M. Pradel, "How transpilation affects quantum code smells: An empirical study," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1–12. [Online]. Available: https://arxiv.org/abs/2308.10318

[72] G. Banti, A. D'Angelo, and M. A. Ferzli, "Antipattern detection in software systems: A systematic review," in *Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, 2017.

[73] P. Oppenheimer and J. Bakos, "Antipatterns in software architecture," *Software: Practice and Experience*, vol. 39, no. 2, pp. 103–125, 2009.

[74] D. Kouzapas and M. Maranzano, "Detecting antipatterns in enterprise systems: A systematic review," *Journal of Software: Evolution and Process*, vol. 32, no. 7, 2020.

[75] R. D. Schwanke and J. W. Smith, "Antipatterns in design and architecture: A literature review," *Journal of Systems and Software*, vol. 168, p. 110634, 2020.

[76] R. Marinescu, "Detection strategies: Detecting design flaws through static analysis," in *Proceedings of the International Conference on Software Maintenance*, 2004.

[77] R. Rani and D. Sharma, "Code smell detection using machine learning: A review," *International Journal of Computer Science and Information Security*, vol. 18, no. 2, pp. 75–82, 2020.

[78] F. Palomba and M. D'Ambros, "Detecting and refactoring code smells: A survey," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 255–274, 2017.

[79] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*. Springer, 2006, vol. 4, no. 4.

[80] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: data mining, inference and prediction*, 2nd ed. Springer, 2009. [Online]. Available: http://www-stat.stanford.edu/~tibs/ElemStatLearn/

[81] S. Parsons, "Introduction to machine learning by ethem alpaydin, mit press, 0-262-01211-1, 400 pp." *Knowledge Eng. Review*, vol. 20, pp. 432–433, 12 2005.

[82] M. Sharma, R. Mittal, A. Bharati, D. Saxena, and A. Singh, "A survey and classification on recommendation systems," in *1*, 01 2022.

[83] J. R. Quinlan, "Induction of decision trees," in *Machine learning*. Springer, 1986, pp. 81–106.

[84] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and regression trees*. Wadsworth International Group, 1984.

[85] A. Liaw and M. Wiener, "Classification and regression by randomforest," *R news*, vol. 2, no. 3, pp. 18–22, 2002.

[86] T. Joachims, "Text categorization with support vector machines: Learning with many relevant features," *Machine learning: ECML-98*, pp. 137–142, 1998.

[87] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[88] B. Schölkopf, A. Smola, and K.-R. Müller, "Input space versus feature space in kernel-based methods," in *IEEE Transactions on Neural Networks*, 1999, pp. 1000–1017.

[89] C.-C. Chang and C.-J. Lin, "Libsvm: A library for support vector machines," *ACM transactions on intelligent systems and technology (TIST)*, vol. 2, no. 3, pp. 1–27, 2011.

[90] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE transactions on information theory*, vol. 13, no. 1, pp. 21–27, 1967.

[91] L. Peterson, "K-nearest neighbor," *Scholarpedia*, vol. 4, p. 1883, 01 2009.

[92] R. Duda, P. Hart, and D. G.Stork, *Pattern Classification*, 01 2001, vol. xx.

[93] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/00031305.1992.10475879

[94] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 06 2014.

[95] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0893608089900208

[96] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[97] J. Friedman, "Greedy function approximation: A gradient boosting machine," *The Annals of Statistics*, vol. 29, 11 2000.

[98] A. Natekin and A. Knoll, "Gradient boosting machines, a tutorial," *Frontiers in Neurorobotics*, vol. 7, 2013. [Online]. Available: https://www.frontiersin.org/journals/neurorobotics/articles/10.3389/fnbot.2013.00021

[99] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 794. [Online]. Available: https://doi.org/10.1145/2939672.2939785

[100] D. D. Lewis, "Naive bayes classifiers," *Encyclopedia of Machine Learning*, pp. 404–407, 1998.

[101] I. Rish, "An empirical comparison of supervised learning algorithms," in *Proceedings of the 2001 International Joint Conference on Artificial Intelligence*, 2001, pp. 152–157.

[102] C. D. Manning, P. Raghavan, and H. Schutze, *Introduction to Information Retrieval*. MIT Press, 2008.

[103] D. R. Cox, "The regression analysis of binary sequences," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 20, no. 2, pp. 215–242, 1958. [Online]. Available: http://www.jstor.org/stable/2983890

[104] D. Hosmer and S. Lemeshow, *Applied Logistic Regression. Hoboken*, 01 2000, vol. 354.

[105] J. Peng, K. Lee, and G. Ingersoll, "An introduction to logistic regression analysis and reporting," *Journal of Educational Research - J EDUC RES*, vol. 96, pp. 3–14, 09 2002.

[106] H. Robbins and S. Monro, "A stochastic approximation method," *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951.

[107] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[108] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of Eugenics*, vol. 7, no. 2, pp. 179–188, 1936.

[109] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, no. 1, pp. 3–42, 2006.

[110] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[111] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2012.

[112] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, 2014.

[113] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

[114] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[115] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[116] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[117] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," in *Neural Computation*, 1999.

[118] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1724–1734.

[119] D. Jude Hemanth and J. Anitha, "Modified genetic algorithm approaches for classification of abnormal magnetic resonance brain tumour images," *Applied Soft Computing*, vol. 75, pp. 21–28, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1568494618306367

[120] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *Proceedings of the Sixth International Symposium on Micro Machine and Human Science (MHS'95)*. IEEE, 1995, pp. 39–43.

[121] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 482. [Online]. Available: https://doi.org/10.1145/3180155.3182532

[122] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment," *Journal of Object Technology*, vol. 15, no. 3, pp. 1–38, 2016.

[123] H. Keshavarz and M. Nagappan, "Apachejit: a large dataset for just-in-time defect prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 195. [Online]. Available: https://doi.org/10.1145/3524842.3527996

[124] M. de Oliveira Barros, F. de Almeida Farzat, and G. H. Travassos, "Learning from optimization: A case study with apache ant," *Information and Software Technology*, vol. 57, pp. 684–704, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584914001839

[125] S. Agrawal and R. Gupta, "Development and comparison of open source based web gis frameworks on wamp and apache tomcat web servers," vol. XL-4, 05 2014.

[126] S. Raemaekers, A. van Deursen, and J. Visser, "The maven repository dataset of metrics, changes, and dependencies," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 221–224.

[127] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.

[128] F. Arcelli Fontana, G. Giai, R. Roveda, and M. Zanoni, "Mlcq: A curated dataset for machine learning-based code quality prediction," *Journal of Systems and Software*, vol. 183, p. 111087, 2022.

[129] R. Shatnawi, "Designite: A software design quality assessment tool," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 49–52.

[130] J. Zhao, "Some size and structure metrics for quantum software," in *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2021, pp. 22–27. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/Q-SE52541.2021.00012

[131] J. Cruz-Lemus, L. Camargo, and M. Piattini, *Towards a Set of Metrics for Quantum Circuits Understandability*. 1, 08 2021, pp. 239–249.

[132] M. Paltenghi and M. Pradel, "LintQ: A Static Analysis Framework for Qiskit Quantum Programs," *1*, 10 2023.

[133] P. Zhao, J. Zhao, Z. Miao, and S. Lan, "Bugs4q: A benchmark of real bugs for quantum programs," 2021.

[134] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, vol. 2, 1995, pp. 1137–1143.

[135] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems*, 2017, pp. 4765–4774.

[136] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should i trust you?" explaining the predictions of any classifier," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 1135–1144.

[137] R. Malhotra, B. Jain, and M. Kessentini, "Examining deep learning's capability to spot code smells: a systematic literature review," *Cluster Computing*, vol. 26, no. 6, pp. 3473–3501, 2023.

[138] O. A. Bastias, J. Diaz, and J. Lopez Fenner, "Exploring the intersection between software maintenance and machine learning ;a systematic mapping study," *Applied Sciences*, vol. 13, no. 3, 2023. [Online]. Available: https://www.mdpi.com/2076-3417/13/3/1710

[139] E. A. AlOmar, A. Ivanov, Z. Kurbatova, Y. Golubev, M. W. Mkaouer, A. Ouni, T. Bryksin, L. Nguyen, A. Kini, and A. Thakur, "Just-in-time code duplicates extraction," *Information and Software Technology*, vol. 158, p. 107169, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058492300023X

[140] Z. Li, S. Wang, W. Wang, P. Liang, R. Mo, and B. Li, "Understanding bugs in multi-language deep learning frameworks," 2023.

[141] J. Gesi, S. Liu, J. Li, I. Ahmed, N. Nagappan, D. Lo, E. S. de Almeida, P. S. Kochhar, and L. Bao, "Code smells in machine learning systems," *arXiv preprint arXiv:2203.00803*, 2022.

[142] S. M. Al Khatib, K. Alkharabsheh, and S. Alawadi, "Selection of human evaluators for design smell detection using dragonfly optimization algorithm: An empirical study," *Information and Software Technology*, vol. 155, p. 107120, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584922002294

[143] D. K. Kim, "A deep neural network-based approach to finding similar code segments," *IEICE TRANSACTIONS on Information and Systems*, vol. 103, no. 4, pp. 874–878, 2020.

[144] K. Alkharabsheh, S. Alawadi, V. R. Kebande, Y. Crespo, M. Fernandez-Delgado, and J. A. Taboada, "A comparison of machine learning algorithms on design smell detection using balanced and imbalanced dataset: A study of god class," *Information and Software Technology*, vol. 143, p. 106736, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921001865

[145] N. Muralidhar, S. Muthiah, P. Butler, M. Jain, Y. Yu, K. Burne, W. Li, D. Jones, P. Arunachalam, H. S. McCormick, and N. Ramakrishnan, "Using antipatterns to avoid mlops mistakes," 2021.

[146] I. Kaur and A. Kaur, "A novel four-way approach designed with ensemble feature selection for code smell detection," *IEEE Access*, vol. 9, pp. 8695–8707, 2021.

[147] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *2018 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2018, pp. 1–12.

[148] B. A. Kitchenham, "Systematic review in software engineering: where we are and where we should be going," in *Proceedings of the 2nd international workshop on Evidential assessment of software technologies*, 2012, pp. 1–2.

[149] H. Gupta, T. G. Kulkarni, L. Kumar, L. B. M. Neti, and A. Krishna, "An empirical study on predictability of software code smell using deep learning models," in *Advanced Information Networking and Applications*. Springer International Publishing, 2021, pp. 120–132. [Online]. Available: https://doi.org/10.1007%2F978-3-030-75075-6_10

[150] A. K. Das, S. Yadav, and S. Dhal, "Detecting code smells using deep learning," in *TENCON 2019 - 2019 IEEE Region 10 Conference (TENCON)*, 2019, p. 2081 2086.

[151] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "On the feasibility of transfer-learning code smells using deep learning," 2019.

[152] H. Nandani, M. Saad, and T. Sharma, "Dacos-a manually annotated dataset of code smells," 2023.

[153] S. Tarwani and A. Chug, "Application of deep learning models for code smell prediction," in *2022 10th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, 2022, pp. 1–5.

[154] H. Liu, Z. Xu, and Y. Zou, "Deep learning based feature envy detection," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, p. 385 396.

[155] M. Zhang and J. Jia, "Feature envy detection with deep learning and snapshot ensemble," in *2022 9th International Conference on Dependable Systems and Their Applications (DSA)*, 2022, pp. 215–223.

[156] H. Wang, J. Liu, J. Kang, W. Yin, H. Sun, and H. Wang, "Feature envy detection based on bi-lstm with self-attention mechanism," in *2020 IEEE Intl Conf on Parallel and Distributed Processing with Applications, Big Data and Cloud Computing, Sustainable Computing and Communications, Social Computing and Networking (ISPA/BDCloud/SocialCom/SustainCom)*, 2020, p. 448 457.

[157] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu, and L. Zhang, "Deep learning based code smell detection," *IEEE transactions on Software Engineering*, vol. 47, no. 9, pp. 1811–1837, 2019.

[158] M. Hadj-Kacem and N. Bouassida, "A hybrid approach to detect code smells using deep learning." in *ENASE*, 2018, pp. 137–146.

[159] M. Garcia-Torres, R. Ruiz, and F. Divina, "Evolutionary feature selection on high dimensional data using a search space reduction approach," *Engineering Applications of Artificial Intelligence*, vol. 117, p. 105556, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0952197622005462

[160] R. Yedida and T. Menzies, "How to improve deep learning for software analytics (a case study with code smell detection)," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 156–166.

[161] A. T. Imam, B. R. Al-Srour, and A. Alhroob, "The automation of the detection of large class bad smell by using genetic algorithm and deep learning," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 6, pp. 2621–2636, 2022.

[162] F. Lomio, S. Moreschini, and V. Lenarduzzi, "A machine and deep learning analysis among sonarqube rules, product, and process metrics for fault prediction," *Empirical Software Engineering*, vol. 27, no. 7, p. 189, 2022.

[163] Y.-B. Jo, J. Lee, and C.-J. Yoo, "Two-pass technique for clone detection and type classification using tree-based convolution neural network," *Applied Sciences*, vol. 11, no. 14, 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/14/6613

[164] W. Xu and X. Zhang, "Multi-granularity code smell detection using deep learning method based on abstract syntax tree," in *1*, vol. 2021-July, Pittsburgh, PA, United states, 2021, pp. 503 – 509. [Online]. Available: http://dx.doi.org/10.18293/SEKE2021-014

[165] S. Kaur and S. Singh, "Improving the quality of open source software," *Agile Software Development: Trends, Challenges and Applications*, pp. 309–323, 2023.

[166] A. Bhave and R. Sinha, "Deep multimodal architecture for detection of long parameter list and switch statements using distilbert," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2022, pp. 116–120.

[167] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice. Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems*. Springer Verlag, 2010.

[168] R. Saheb-Nassagh, M. Ashtiani, and B. Minaei-Bidgoli, "A probabilistic-based approach for automatic identification and refactoring of software code smells," *Applied Soft Computing*, vol. 130, p. 109658, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1568494622007074

[169] F. Li, K. Zou, J. W. Keung, X. Yu, S. Feng, and Y. Xiao, "On the relative value of imbalanced learning for code smell detection," *Software: Practice and Experience*, vol. 53, no. 10, pp. 1902–1927, 2023.

[170] Y. M. B. Ali, "Adversarial attacks on deep learning networks in image classification based on smell bees optimization algorithm," *Future Generation Computer Systems*, vol. 140, pp. 185–195, 2023.

[171] R. Gupta and S. Kumar Singh, "A novel metric based detection of temporary field code smell and its empirical analysis," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 10, Part B, pp. 9478–9500, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157821003050

[172] I. Shoenberger, M. W. Mkaouer, and M. Kessentini, "On the use of smelly examples to detect code smells in javascript," in *Applications of Evolutionary Computation*, G. Squillero and K. Sim, Eds. Cham: Springer International Publishing, 2017, pp. 20–34.

[173] F. Palomba, A. Zaidman, G. Bavota, R. Oliveto, and A. De Lucia, "A replicated study on the use of history-based metrics for defect prediction," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 491–495.

[174] S. Tummalapalli, L. Kumar, and N. L. Bhanu Murthy, "Web service anti-patterns detection using cnn with varying sequence padding size," in *Mobile Application Development: Practice and Experience: 12th Industry Symposium in Conjunction with 18th ICDCIT 2022*. Springer, 2023, pp. 153–165.

[175] R. Sepahvand, R. Akbari, B. Jamasb, S. Hashemi, and O. Boushehrian, "Using word embedding and convolution neural network for bug triaging by considering design flaws," *Sci. Comput. Program.*, vol. 228, no. C, jun 2023. [Online]. Available: https://doi.org/10.1016/j.scico.2023.102945

[176] J. Virmajoki, A. Knutas, and J. Kasurinen, "Detecting code smells with ai: a prototype study," in *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*, 2022, pp. 1393–1398.

[177] W. Hua, Y. Sui, Y. Wan, G. Liu, and G. Xu, "Fcca: Hybrid code representation for functional clone detection using attention networks," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 304–318, 2021.

[178] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "Code smell detection by deep direct-learning and transfer-learning," *Journal of Systems and Software*, vol. 176, p. 110936, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221000339

[179] A. Barbez, F. Khomh, and Y.-G. Gueheneuc, "Deep learning anti-patterns from code metrics history," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 114–124.

[180] D. K. Kim, "Finding bad code smells with neural network models," *International Journal of Electrical and Computer Engineering*, vol. 7, no. 6, p. 3613, 2017.

[181] R. Malhotra, B. Jain, and M. Kessentini, "Dir-smote: a density-influence resampling framework for imbalanced code smell detection," *Automated Software Engineering*, vol. 33, no. 3, p. 76, 2026.

[182] P. S. Yadav, R. S. Rao, and A. Mishra, "An evaluation of multi-label classification approaches for method-level code smells detection," *IEEE Access*, 2024.

[183] Y. Zhang, C. Ge, H. Liu, and K. Zheng, "Code smell detection based on supervised learning models: A survey," *Neurocomputing*, vol. 565, p. 127014, 2024.

[184] A. Kovacevic, N. Luburic, J. Slivka, S. Prokic, K. G. Grujic, D. Vidakovic, and G. Sladic, "Automatic detection of code smells using metrics and codet5 embeddings: a case study in c#," *Neural Computing and Applications*, vol. 36, no. 16, p. 9220, 2024.

[185] N. Bafatakis, N. Boecker, W. Boon, M. C. Salazar, J. Krinke, G. Oznacar, and R. White, "Python coding style compliance on stack overflow," in *1*, ser. MSR '19. IEEE Press, 2019, p. 2. [Online]. Available: https://doi.org/10.1109/MSR.2019.00042

[186] P. S. Thakur, M. Jadeja, and S. S. Chouhan, "Enhancing software code smell detection with modified cost-sensitive svm," *International Journal of System Assurance Engineering and Management*, p. 15, 2024.

[187] P. S. Thakur, M. Jadeja, and S. S. Chouhan, "Cbret: A cluster-based resampling technique for dealing with imbalanced data in code smell prediction," *Knowledge-Based Systems*, vol. 286, p. 111390, 2024.

[188] D. Elreedy, A. F. Atiya, and F. Kamalov, "A theoretical distribution analysis of synthetic minority oversampling technique (smote) for imbalanced learning," *Machine Learning*, vol. 113, no. 7, pp. 4903–4923, 2024.

[189] Y. Bao and S. Yang, "Two novel smote methods for solving imbalanced classification problems," *IEEE Access*, vol. 11, pp. 5816–5823, 2023.

[190] S. K. Satapathy, S. Mishra, P. K. Mallick, and G.-S. Chae, "Adasyn and abc-optimized rbf convergence network for classification of electroencephalograph signal," *Personal and Ubiquitous Computing*, pp. 1–17, 2023.

[191] R. Mitra, A. Bajpai, and K. Biswas, "Adasyn-assisted machine learning for phase prediction of high entropy carbides," *Computational Materials Science*, vol. 223, p. 112142, 2023.

[192] Z. Salekshahrezaee, J. L. Leevy, and T. M. Khoshgoftaar, "The effect of feature extraction and data sampling on credit card fraud detection," *Journal of Big Data*, vol. 10, no. 1, p. 6, 2023.

[193] H. Hairani, A. Anggrawan, and D. Priyanto, "Improvement performance of the random forest method on unbalanced diabetes data classification using smote-tomek

link," *JOIV: international journal on informatics visualization*, vol. 7, no. 1, pp. 258–264, 2023.

[194] Z. Teng, P. Cao, M. Huang, Z. Gao, and X. Wang, "Multi-label borderline oversampling technique," *Pattern Recognition*, vol. 145, p. 109953, 2024.

[195] B. H. van der Velden, H. J. Kuijf, K. G. Gilhuijs, and M. A. Viergever, "Explainable artificial intelligence (xai) in deep learning-based medical image analysis," *Medical Image Analysis*, vol. 79, p. 102470, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1361841522001177

[196] X. Wang, J. Ren, H. Ren, W. Song, Y. Qiao, Y. Zhao, L. Linghu, Y. Cui, Z. Zhao, L. Chen *et al.*, "Diabetes mellitus early warning and factor analysis using ensemble bayesian networks with smote-enn and boruta," *Scientific Reports*, vol. 13, no. 1, p. 12718, 2023.

[197] P. S. Yadav, R. S. Rao, A. Mishra, and M. Gupta, "Machine learning-based methods for code smell detection: A survey," *Applied Sciences*, vol. 14, no. 14, 2024. [Online]. Available: https://www.mdpi.com/2076-3417/14/14/6149

[198] D. Salerno, "Edtforcsd," n.d., accessed: 2024-09-18. [Online]. Available: https://github.com/DanieleSalerno/EDTforCSD

[199] X. Li, M. Du, J. Chen, Y. Chai, H. Lakkaraju, and H. Xiong, "A unified xai benchmark for faithfulness evaluation of feature attribution methods across metrics, modalities and models," *Advances in Neural Information Processing Systems*, vol. 36, pp. 1630–1643, 2023.

[200] D. Wu, F. Mu, L. Shi, Z. Guo, K. Liu, W. Zhuang, Y. Zhong, and L. Zhang, "ismell: Assembling llms with expert toolsets for code smell detection and refactoring," ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 1345–1357. [Online]. Available: https://doi.org/10.1145/3691620.3695508

[201] A. Barbez, F. Khomh, and Y.-G. Gueheeuc, "A machine-learning based ensemble method for anti-patterns detection," *Journal of Systems and Software*, vol. 161, p. 110486, 2020.

[202] H. S. Munir, S. Ren, M. Mustafa, C. N. Siddique, and S. Qayyum, "Attention based gru-lstm for software defect prediction," *Plos one*, vol. 16, no. 3, p. e0247444, 2021.

[203] S. Dewangan, R. S. Rao, A. Mishra, and M. Gupta, "A novel approach for code smell detection: an empirical study," *IEEE Access*, vol. 9, pp. 162 869–162 883, 2021.

[204] R. Malhotra, B. Jain, and M. Kessentini, "An empirical study on feature extraction using ensemble deep learning for code smell detection," in *International Conference on Business Intelligence and Data Analytics*. Springer, 2024, pp. 573–585.

[205] M. Zhang, J. Jia, L. F. Capretz, X. Hou, and H. Tan, "Graph neural network-based long method and blob code smell detection," *Science of Computer Programming*, vol. 243, p. 103284, 2025.

[206] J. Zeng *et al.*, "Learning graph-based code representations for source-level functional analysis," in *International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2023, pp. 123–134.

[207] S. McIntosh, A. E. Hassan, and J. Rilling, "Mlcq: A large-scale industry-relevant dataset for code smell detection," *Empirical Software Engineering*, vol. 29, no. 1, pp. 1–23, 2024.

[208] Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2017.

[209] A. Alazba, H. Aljamaan, and M. Alshayeb, "Deep learning approaches for bad smell detection: a systematic literature review," *Empirical Software Engineering*, vol. 28, no. 3, p. 77, 2023.

[210] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations (ICLR)*, 2018.

[211] S. Wang, J. Zhang, X. Li, and L. Gong, "Detecting software vulnerabilities using deep learning-based code property graphs," *Journal of Systems and Software*, vol. 170, p. 110798, 2020.

[212] A. Sharma and D. Spinellis, "Application of deep learning for code smell detection: A systematic literature review," *SN Computer Science*, vol. 5, no. 6, pp. 1–15, 2024.

[213] H. Aljamaan, "Voting heterogeneous ensemble for code smell detection," in *2021 20th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2021, pp. 897–902.

[214] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[215] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[216] M. Benedetti, J. Realpe-Gómez, R. Biswas, and A. Perdomo-Ortiz, "Estimation of effective temperatures in quantum annealers for sampling applications: A case study with possible applications in deep learning," *Physical Review A*, vol. 94, no. 2, p. 022308, 2016.

[217] R. Barends, J. Kelly, A. Megrant, A. Veitia, D. Sank, E. Jeffrey, T. C. White, J. Mutus, A. G. Fowler, B. Campbell *et al.*, "Superconducting quantum circuits at the surface code threshold for fault tolerance," *Nature*, vol. 508, no. 7497, pp. 500–503, 2014.

[218] C. H. Bennett and G. Brassard, "Quantum cryptography: Public key distribution and coin tossing," *arXiv preprint arXiv:2003.06557*, 2020.

[219] M. A. Serrano, J. A. Cruz-Lemus, R. Perez-Castillo, and M. Piattini, "Quantum software components and platforms: Overview and quality assessment," *ACM Comput. Surv.*, vol. 55, no. 8, dec 2022. [Online]. Available: https://doi.org/10.1145/3548679

[220] J. Zhao, "Quantum software engineering: Landscapes and horizons," *arXiv preprint arXiv:2007.07047*, 2020.

[221] R. Shaydulin, C. Thomas, and P. Rodeghero, "Making quantum computing open: Lessons from open source projects," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 451–455.

[222] H. Ball, M. J. Biercuk, A. R. R. Carvalho, J. Chen, M. Hush, L. A. D. Castro, L. Li, P. J. Liebermann, H. J. Slatyer, C. Edmunds, V. Frey, C. Hempel, and A. Milne, "Software tools for quantum control: improving quantum computer performance through noise and error suppression," *Quantum Science and Technology*, vol. 6, no. 4, p. 044011, sep 2021. [Online]. Available: https://dx.doi.org/10.1088/2058-9565/abdca6

[223] F. Leymann, J. Barzen, M. Falkenthal, D. Vietz, B. Weder, and K. Wild, "Quantum in the cloud: Application potentials and research opportunities," 2020.

[224] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie, "Projection-based runtime assertions for testing and debugging quantum programs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–29, 2020.

[225] S. Ali, P. Arcaini, X. Wang, and T. Yue, "Assessing the effectiveness of input and output coverage criteria for testing quantum programs," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2021, pp. 13–23.

[226] I. Ahmed, C. Brindescu, U. A. Mannan, C. Jensen, and A. Sarma, "An empirical examination of the relationship between code smells and merge conflicts," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, pp. 58–67.

[227] P. Zhao, J. Zhao, and L. Ma, "Identifying bug patterns in quantum programs," in *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*. IEEE, 2021, pp. 16–21.

[228] H. Jebnoun, H. Ben Braiek, M. M. Rahman, and F. Khomh, "The scent of deep learning code: An empirical study," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 420–430.

[229] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 682–691.

[230] M. Agnihotri and A. Chug, "A systematic literature survey of software metrics, code smells and refactoring techniques," *Journal of Information Processing Systems*, vol. 16, no. 4, pp. 915–934, 2020.

[231] M. Y. Mhawish and M. Gupta, "Predicting code smells and analysis of predictions: using machine learning techniques and software metrics," *Journal of Computer Science and Technology*, vol. 35, no. 6, pp. 1428–1445, 2020.

[232] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE transactions on systems, man, and cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[233] E. Santos and H. Gomes, "A comparative study of polynomial kernel svm applied to appearance-based object recognition," in *1*, 07 2002, pp. 408–418.

[234] H.-T. Lin and C.-J. Lin, "A study on sigmoid kernels for svm and the training of non-psd kernels by smo-type methods," *Neural Computation*, 06 2003.

[235] X. Ding, J. Liu, F. Yang, and J. Cao, "Random radial basis function kernel-based support vector machine," *Journal of the Franklin Institute*, vol. 358, 10 2021.

[236] P. Domingos and M. Pazzani, "Beyond independence: Conditions for the optimality of the simple bayesian classi er," in *Proc. 13th Intl. Conf. Machine Learning*, 1996, pp. 105–112.

[237] V. Anand, B. KiranBala, S. Srividhya, K. C., M. Younus, and M. H. Rahman, "Gaussian naive bayes algorithm: A reliable technique involved in the assortment of the segregation in cancer," *Mobile Information Systems*, vol. 2022, pp. 1–7, 06 2022.

[238] M. Abbas, K. Ali, S. Memon, A. Jamali, S. Memon, and A. Ahmed, "Multinomial naive bayes classification model for sentiment analysis," *1*, 03 2019.

[239] C. Dawson and R. Wilby, "Hydrological modelling using artificial neural networks," *Progress in physical Geography*, vol. 25, no. 1, pp. 80–108, 2001.

[240] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*. Springer, 2010, pp. 177–186.

[241] B. Loef, A. Wong, N. Janssen, M. Strak, J. Hoekstra, S. Picavet, H. Boshuizen, W. Verschuren, and G.-C. Herber, "Using random forest to identify longitudinal predictors of health in a 30-year cohort study," *Scientific Reports*, vol. 12, p. 10372, 06 2022.

[242] K. Fukunaga, *Introduction to statistical pattern recognition*. Elsevier, 2013.

[243] Z. Hanusz, J. Tarasinska, and W. Zielinski, "Shapiro wilk test with known mean," *Revstat Statistical Journal*, vol. 14, pp. 89–100, 02 2016.

[244] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric statistical methods*. John Wiley & Sons, 2013.

[245] J. Demsar, "Statistical comparisons of classifiers over multiple data sets," *J. Mach. Learn. Res.*, vol. 7, p. 30, dec 2006.

[246] R. Malhotra, B. Jain, and M. Kessentini, "Qnet: exploring deep learning for quantum code smell detection," *Software Quality Journal*, vol. 33, no. 1, p. 25, 2025. [Online]. Available: https://doi.org/10.1007/s11219-025-09724-5

[247] K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra, "Investigating the effect of coupling metrics on fault proneness in object-oriented systems," *Software Quality Professional*, vol. 8, no. 4, p. 4, 2006.

[248] F. Arcelli Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowledge-Based Systems*, vol. 128, pp. 43–58, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950705117301880

[249] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 93–104.

[250] T. Lewowski and L. Madeyski, *Code Smells Detection Using Artificial Intelligence Techniques: A Business-Driven Systematic Review*. Cham: Springer International Publishing, 2022, ch. SSDC,volume 377, pp. 285–319. [Online]. Available: https://doi.org/10.1007/978-3-030-77916-0_12

[251] Y. Singh, A. Kaur, and R. Malhotra, "Predicting testing effort using artificial neural network," in *1*, 2023.

[252] D. J. Montana and L. Davis, "Training feedforward neural networks using genetic algorithms," in *International Joint Conference on Artificial Intelligence*, 1989. [Online]. Available: https://api.semanticscholar.org/CorpusID:6336712

[253] D. Soto and W. Soto, "Evolutionary algorithm for solving supervised classification problems: An experimental study," in *Proceedings of the 2022 6th International Conference on Intelligent Systems, Metaheuristics & Swarm Intelligence*, ser. ISMSI 22. New York, NY, USA: Association for Computing Machinery, 2022, p. 29. [Online]. Available: https://doi.org/10.1145/3533050.3533054

[254] Y.-H. Shih and C.-K. Ting, "Evolutionary optimization on k-nearest neighbors classifier for imbalanced datasets," in *2019 IEEE Congress on Evolutionary Computation (CEC)*, 2019, pp. 3348–3355.

[255] I. Revin, V. A. Potemkin, N. R. Balabanov, and N. O. Nikitin, "Automated machine learning approach for time series classification pipelines using evolutionary optimization," *Knowledge-Based Systems*, vol. 268, p. 110483, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950705123002332

[256] J. H. Joloudari, H. Saadatfar, A. Dehzangi, and S. Shamshirband, "Computer-aided decision-making for predicting liver disease using pso-based optimized svm with feature selection," *Informatics in medicine unlocked*, vol. 17, p. 100255, 2019.

[257] L. Madeyski and T. Lewowski, "Mlcq: Industry-relevant code smell data set," in *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 347. [Online]. Available: https://doi.org/10.1145/3383219.3383264

[258] A. Kovacevic, J. Slivka, D. Vidakovic, K.-G. Grujic, N. Luburic, S. Prokic, and G. Sladic, "Automatic detection of long method and god class code smells through neural source code embeddings," *Expert Systems with Applications*, vol. 204, p. 117607, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417422009186

[259] N. Alomari, A. Alazba, H. Aljamaan, and M. Alshayeb, "Smellycode++: Multi-label dataset for code smell detection," *Scientific Data*, vol. 12, no. 1, p. 1207, 2025.

[260] A. Sharma, P. Kaur, and R. Singh, "Multi-language code smells instances: A cross-lingual dataset," *Information and Software Technology*, vol. 170, p. 107452, 2024.

[261] M. D. Stefano, D. D. Nucci, F. Palomba, and A. D. Lucia, "An empirical study into the effects of transpilation on quantum circuit smells," *Empirical Software Engineering*, vol. 29, no. 3, p. 61, 2024.

[262] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," 2020. [Online]. Available: https://arxiv.org/abs/1905.11946

[263] F. Fontana, P. Martín, S. Mariani, and M. Zanoni, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[264] D. Di Nucci, F. Palomba, G. Bavota, R. Oliveto, and D. Poshyvanyk, "A developer-centered bug prediction model," *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 463–486, 2018.

[265] G. Saranya, H. K. Nehemiah, and A. Kannan, "Hybrid particle swarm optimisation with mutation for code smell detection," *International Journal of Bio-Inspired Computation*, vol. 12, no. 3, pp. 186–195, 2018.

[266] U. Draz, A. Alahmadi, K. Alfakeeh, and A. Alharbi, "Code smell detection using whale optimization algorithm," *Computers, Materials & Continua*, vol. 68, no. 2, pp. 1963–1980, 2021.

[267] R. Malhotra, B. Jain, and M. Kessentini, "A comparative analysis of quantum programming bug detection tools," in *International Conference on Data Science and Applications*. Springer, 2024, pp. 115–127.

[268] S. Honarvar, M. Mousavi, and R. Nagarajan, "Property-based testing of quantum programs in q#," in *1*, 06 2020, pp. 430–435.

[269] S. Johnstun and J.-F. Van Huele, "Understanding and compensating for noise on IBM quantum computers," *American Journal of Physics*, vol. 89, no. 10, pp. 935–942, 10 2021. [Online]. Available: https://doi.org/10.1119/10.0006204

[270] D. Fortunato, J. CAMPOS, and R. ABREU, "Mutation testing of quantum programs: A case study with qiskit," *IEEE Transactions on Quantum Engineering*, vol. 3, pp. 1–17, 2022.

[271] P. Mateus, A. Sernadas, and A. Souto, "Universality of quantum Turing machines with deterministic control," *Journal of Logic and Computation*, vol. 27, no. 1, pp. 1–19, 02 2015. [Online]. Available: https://doi.org/10.1093/logcom/exv008

[272] X. Wang, P. Arcaini, T. Yue, and S. Ali, "Quito: a coverage-guided test generator for quantum programs," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1237–1241.

[273] Google Quantum AI, "Suppressing quantum errors by scaling a surface code logical qubit," *Nature*, vol. 614, no. 7949, pp. 676–681, 2023. [Online]. Available: https://doi.org/10.1038/s41586-022-05434-1

[274] P. Zhao, X. Wu, Z. Li, and J. Zhao, "Qchecker: Detecting bugs in quantum programs via static analysis," 2023.

[275] P. K. Nayak, K. V. Kher, M. B. Chandra, M. V. P. Rao, and L. Zhang, "Q-pac: Automated detection of quantum bug-fix patterns," 2023.

[276] M. Hadj-Kacem and N. Bouassida, "Deep representation learning for code smells detection using variational auto-encoder," in *2019 International Joint Conference on Neural Networks (IJCNN)*, 2019, pp. 1–8.

[277] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, p. 249 260.

[278] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 87 98. [Online]. Available: https://doi.org/10.1145/2970276.2970326

[279] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, sep 2014. [Online]. Available: https://doi.org/10.1145/2629648