

Adaptive Online Learning for Software Defect Prediction with Defective Overlooking Consideration

*Dissertation submitted in partial fulfillment of the requirement
for the award of the degree of*

MASTER OF COMPUTER APPLICATIONS

By

Vanama Nikhith

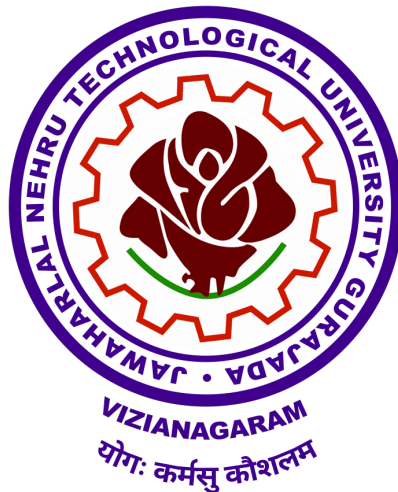
23VV1F0027

Under the Esteemed Guidance of

Dr. B. Tirimula Rao, M.Tech, Ph.D.

Assistant Professor

Department of Information Technology



DEPARTMENT OF INFORMATION TECHNOLOGY

JNTUGV College of Engineering Vizianagaram (Autonomous)

Jawaharlal Nehru Technological University, Gurajada Vizianagaram

Dwarapudi, Vizianagaram-535003, Andhra Pradesh, India

2024-2025

DEPARTMENT OF INFORMATION TECHNOLOGY
JNTUGV COLLEGE OF ENGINEERING
VIZIANAGARAM



Certificate

This is to certify that the Dissertation report entitled [Adaptive Online Learning for Software Defect Prediction with Defective Overlooking Consideration](#), that is being submitted by Vanama Nikhith bearing registration number (23VV1F0027) in partial fulfillment for the degree of [Master of Computer Applications \(MCA\)](#) from Jawaharlal Nehru Technological University Gurajada Vizianagaram - College of Engineering Vizianagaram. This bonafide work was carried out by him under my guidance and supervision during the year 2024 - 2025.

The result embodied in this report has not been submitted to any other University or Institute for the award of any degree or diploma.

Signature of Project Guide

Dr. B. TIRIMULA RAO

Assistant Professor

Dept. of Information Technology

Signature of Head of the Department

Dr. Ch. Bindu Madhuri

Assistant Professor & HOD

Dept. of Information Technology

(Signature of External Examiner)

DECLARATION

I **Vanama Nikhith (Reg. No: 23VV1F0027)**, declare that this written submission represents my ideas in my own words and where others' ideas or words have been included. I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea data fact source in my submission. I understand that any violation of the above will be cause for disciplinary action by the institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

(Signature)

V.Nikhith

(23VV1F0027)

Date :

Place :



DEPARTMENT OF INFORMATION AND TECHNOLOGY
JNTU-GV College of Engineering Vizianagaram (Autonomous)
Jawaharlal Nehru Technological University, Gurajada Vizianagaram
Dwarapudi, Vizianagaram - 535003, Andhra Pradesh, India
2024 - 2025

Website : www.jntukucev.ac.in

Subject Name : Dissertation

Regulation : R20

Subject Code : MCA4103

Academic Year : 2025

CO'S

Course Outcomes

Course Outcomes	
CO1	To develop the work practice in students to apply theoretical and practical tools/techniques to solve real life problems related to industry and current research.
CO2	Complete an independent research project, resulting in at least a thesis publication and research outputs in terms of publications in high impact factor journals, conference proceedings and patents.
CO3	Demonstrate knowledge of contemporary issues in their chosen field of research.
CO4	Demonstrate an ability to present and defend their research work to panel of experts.
CO5	Be able to apply the knowledge of computing tools and techniques in the selected field for solving real world problems encounter in the software industries.

CO-PO Mapping

Mapping of Course Outcomes (COs) with Program Outcomes (POs)

Course Out Comes	Program Outcomes (POs)														
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
CO1	3	3	1	2	2	-	-	-	3	3	2	3	3	3	2
CO2	3	3	2	2	2	-	-	-	2	3	3	2	3	2	2
CO3	3	3	2	1	3	-	-	1	2	2	3	3	3	3	2
CO4	2	3	1	2	2	-	-	1	2	3	2	2	2	3	1
CO5	3	2	2	3	1	-	-	-	2	3	1	3	1	2	1

Enter correlation levels 1,2 and 3 as given below:

1.Slight(low) 2.Moderate(Medium) 3.Substantial(High) If there no correlation put “-”

Signature of the Student

Signature of the Guide

Signature of the HOD

ACKNOWLEDGEMENT

This acknowledgment transcends the reality of formality when I express deep gratitude and respect to all those people behind the screen who inspired and helped us in the completion of this project work.

I take the privilege to express my heartfelt gratitude to my guide **Dr. B. TIRIMULA RAO**, Assistant Professor , Department of Information Technology, JNTUGV-CEV for his valuable suggestions and constant motivation that greatly helped me in the successful completion of the project. Wholehearted cooperation and the keen interest shown by him at all stages are beyond words of gratitude.

With great pleasure and privilege, I wish to express my heartfelt sense of gratitude and indebtedness to **Dr. Ch. Bindu Madhuri**, Assistant Professor, Head of the Department, JNTUGV-CEV, for her supervision.

I express my sincere thanks to Project Coordinator **Pynam Venkateswarlu**, Assistant Professor(C) Department of Information Technology, JNTU-GURAJADA VIZIANAGARAM for his continuous support.

I extend heartfelt thanks to our principal **Prof. R. Rajeswara Rao** for providing intensive support throughout my project.

I am also thankful to all the Teaching and Non-Teaching staff of the Information Technology Department, JNTUGV-CEV, for their direct and indirect help provided to me in completing the project.

I extend my thanks to my parents and friends for their help and encouragement in the success of my project

V.Nikhith
(23VV1F0027)

ABSTRACT

Name of the student: **Vanama Nikhith**

Roll No: **23VV1F0027**

Degree for which submitted: **M.C.A**

Department: **INFORMATION TECHNOLOGY**

Project Title: **Adaptive Online Learning for Software Defective Prediction with Defective Overlooking Consideration**

Project Guide : **Dr. B. TIRIMULA RAO**

Month and Year of thesis submission: **July**

Software defect prediction (SDP) plays a vital role in ensuring software quality by identifying defect-prone modules early in the development lifecycle. Traditional machine learning models typically operate in offline settings, assuming static data distributions. However, in real-world software projects, the code base and defect patterns evolve continuously, leading to issues such as concept drift and the potential overlooking of critical defects.

This research presents an online adaptive learning framework for software defect prediction that leverages the Random Forest algorithm in an incremental setting. The framework is designed to address the challenge of overlooking defects, where some defects remain undetected due to changing project requirements or data inconsistencies.

To further enhance predictive performance, the framework incorporates cost-sensitive learning to tackle class imbalance, a common issue in defect datasets. Experimental evaluations using real-world SDP datasets demonstrate that the adaptive Random Forest approach significantly improves defect detection metrics, particularly precision, recall, and the F1 score, compared to static models.

By integrating incremental updates and addressing overlooked defects, this work contributes to the development of robust, scalable, and real-time defect prediction models for dynamic software engineering environments.

Contents

Acknowledgements	v
Abstract	vi
List of Figures	ix
1 Introduction	1
1.1 Adaptive Online Learning for Software Defect Prediction with Defective Overlooking Consideration	1
1.1.1 Working of Adaptive Online Learning with Defective Overlooking Consideration	2
1.1.2 Advantages and Challenges of Adaptive Online Learning with Defective Overlooking Consideration	3
1.2 SMOTE Analysis	4
1.2.1 Class Imbalance in Software Defect Prediction	4
1.2.2 Balanced vs. Imbalanced Data	4
1.2.3 Synthetic Minority Over-sampling Technique (SMOTE)	5
1.2.4 Role of SMOTE in This Research	5
1.2.5 Evaluation of SMOTE Effectiveness	6
2 Literature Review	7
2.1 Software Defect Prediction	7
2.2 Adaptive Online Learning in Software Defect Prediction	7
2.3 Defective Overlooking in Online Learning	8
2.4 Class Imbalance Problem in Software Defect Prediction	9
2.5 SMOTE-Based Approaches in Software Defect Prediction	9
2.6 Summary	10
3 Software and Hardware Requirements	11
3.1 Software Requirements	11
3.2 Hardware Requirements	12
4 Dataset	13
4.1 Data Collection	13
4.2 Source	14
4.3 Dataset Description	14

5	Architecture	16
5.1	Phase 1: Data Pre-processing	17
5.2	Phase 2: Adaptive Model Learning and Evaluation	17
5.3	SMOTE Analysis and Working	18
5.4	Summary	19
6	Methodology:	20
6.1	Data Collection	20
6.2	Data Preprocessing	20
6.3	Feature Extraction	21
6.4	Online Adaptive Learning Strategy	21
6.5	Defective Overlooking Consideration	22
6.6	Model Training and Evaluation	22
6.7	Performance Metrics	22
7	Implementation	24
7.1	Data Collection	24
7.2	Data Preprocessing	24
7.2.1	SMOTE Analysis	25
7.3	Feature Selection	27
7.4	Adaptive Model Definition and Training	27
7.4.1	Adaptive Learning Approach	27
7.4.2	Online Classifiers Used	27
7.4.3	Defective Overlooking Handling	28
7.5	Performance Metrics	28
7.5.1	Evaluation Strategy	29
7.6	Conclusion	29
8	Results and Analysis	30
8.1	Dataset Description	30
8.2	Preprocessing and Balancing	30
8.3	Modeling with Random Forest	31
8.4	Performance Evaluation	31
8.4.1	Confusion Matrix Visualization	31
8.5	Discussion	32
8.6	Performance Comparison	32
9	Conclusion	35
9.1	Future Scope	36

List of Figures

1.1	Workflow of Adaptive Online Learning with Defective Overlooking Consideration	2
4.1	NASA PROMISE Software Defect Datasets	15
5.1	System Architecture	16
7.1	Dataset Information	24
7.2	Before SMOTE Analysis	26
7.3	After SMOTE Analysis	26
8.1	Performance Evaluation	31
8.2	Confusion Matrix - Random Forest Classifier	32
8.3	Accuracy Comparison	34

Abbreviations

ML Machine Learning

OL Online Learning

SDP Software Defect Prediction

TP True Positive

TN True Negative

FP False Positive

FN False Negative

FP-Rate False Positive Rate

TP-Rate True Positive Rate

F1-Score Harmonic Mean of Precision and Recall

CM Confusion Matrix

SVM Support Vector Machine

Chapter 1

Introduction

1.1 Adaptive Online Learning for Software Defect Prediction with Defective Overlooking Consideration

Machine Learning (ML) is a pivotal area of Artificial Intelligence that enables systems to learn from data and make predictions or decisions without explicit programming. It has revolutionized many fields such as finance, healthcare, transportation, and software engineering. Within the software development lifecycle, the ability to predict and prevent software defects is crucial to ensure product quality, reduce maintenance costs, and improve reliability.

Software Defect Prediction (SDP) applies ML algorithms on historical software data — including code complexity, change history, and defect records — to predict whether new software modules are likely to contain defects. This predictive insight helps prioritize quality assurance and testing efforts effectively.

To overcome this, **adaptive online learning** models update incrementally as new data arrives, allowing the model to learn continuously and adapt to changes such as concept drift — shifts in the underlying data distribution caused by evolving development practices.

A significant challenge in online SDP is **defective overlooking**, where defective instances are misclassified or ignored during incremental updates. This results in missed defect detection, decreasing the model's utility in practice.

This research proposes an adaptive online learning framework that explicitly incorporates mechanisms to detect and mitigate defective overlooking. By combining concept drift detection, incremental updates, and defect re-evaluation strategies, the framework maintains high defect detection accuracy in dynamic software environments.

The framework is evaluated on real-world open-source project datasets and compared against traditional offline methods. Experimental results demonstrate superior performance and robustness of the proposed adaptive model, particularly in reducing overlooked defects over time.

1.1.1 Working of Adaptive Online Learning with Defective Overlooking Consideration

Adaptive online learning continuously updates the predictive model with each new software metric or defect report, unlike batch learning which trains once on a fixed dataset. This incremental update enables the model to stay current as software evolves.

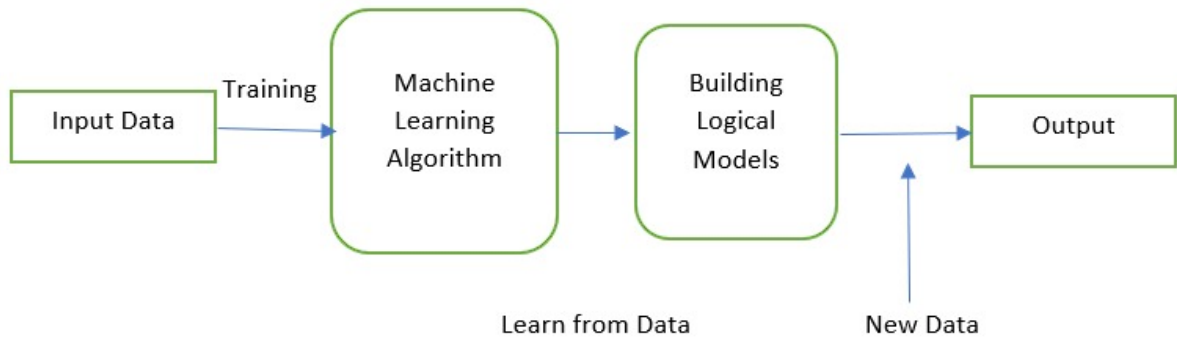


FIGURE 1.1: Workflow of Adaptive Online Learning with Defective Overlooking Consideration

The cycle begins with data collection of software metrics and defect labels. As new commits occur, the model incrementally learns from them. Concept drift detection monitors for changes in data patterns and triggers model adaptation. Crucially, the system implements defective overlooking management to identify defective instances that were misclassified in prior iterations and re-incorporate them, improving recall of defective modules.

This ongoing process of data integration, model update, monitoring, and defect re-evaluation helps maintain accuracy and responsiveness in changing software development contexts.

1.1.2 Advantages and Challenges of Adaptive Online Learning with Defective Overlooking Consideration

Advantages:

- **Real-Time Adaptation:** Continuous learning avoids costly retraining and quickly adapts to new defect patterns.
- **Improved Defect Detection:** Mitigation of defective overlooking reduces missed defects and improves overall recall.
- **Scalability:** Incremental updates handle large, growing datasets efficiently.
- **Reduced Downtime:** Online updating ensures the model remains accurate and avoids degradation over time.
- **Proactive Quality Assurance:** Early detection enables quicker fixes and better resource allocation.

Challenges:

- **Defective Overlooking:** Identifying overlooked defects in incremental updates is complex.
- **Concept Drift Handling:** Effective detection and adaptation to changing data distributions is critical.

- **Noise Sensitivity:** Models may overfit transient anomalies without careful regularization.
- **Model Stability:** Frequent updates risk instability if hyperparameters are not well tuned.
- **Continuous Evaluation:** Streaming evaluation metrics complicate performance monitoring.

This work addresses these challenges through a novel adaptive online learning framework designed for practical software defect prediction.

1.2 SMOTE Analysis

1.2.1 Class Imbalance in Software Defect Prediction

Software defect datasets often suffer from a severe class imbalance problem, where the number of defective modules (minority class) is much smaller than the number of clean modules (majority class). For example, in many projects, defective instances may constitute less than 10% of the dataset.

This imbalance biases standard machine learning models towards the majority class, leading to high accuracy but poor detection of defective modules. Since the minority defective class is typically of greater interest, handling class imbalance is crucial for effective SDP.

1.2.2 Balanced vs. Imbalanced Data

Imbalanced Data: When the minority class (defective) is underrepresented, models trained on such data tend to predict the majority class (non-defective) more often, resulting in low recall and high false negatives for defects. This undermines the purpose of SDP, which aims to flag potential defects early.

Balanced Data: In a balanced dataset, the numbers of defective and non-defective samples are roughly equal. Balanced data help models learn the minority class patterns more effectively, improving recall and reducing overlooking of defects.

1.2.3 Synthetic Minority Over-sampling Technique (SMOTE)

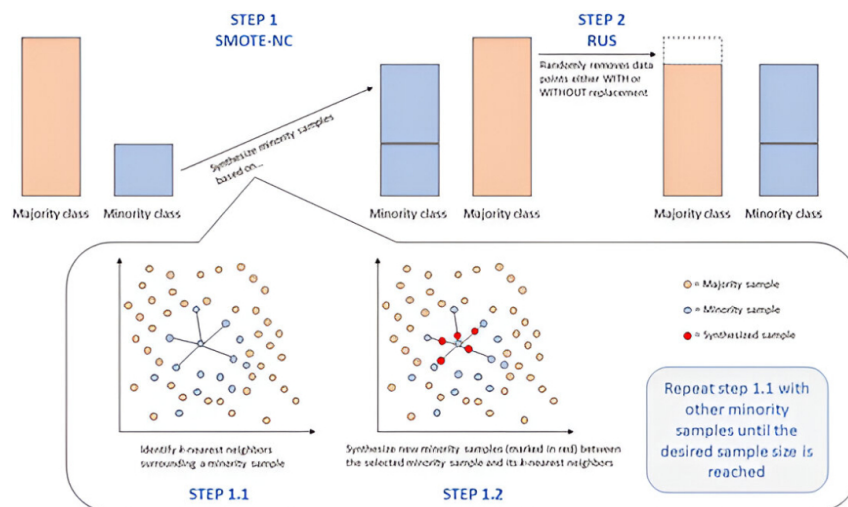
SMOTE is a popular data-level approach to address class imbalance. Unlike naive over-sampling which duplicates minority samples, SMOTE generates synthetic minority class examples by interpolating between existing minority instances and their nearest neighbors.

Specifically, for each minority class sample, SMOTE selects one or more nearest neighbors and creates synthetic samples along the line segments joining the minority sample and its neighbors. This generates new, plausible minority examples, enriching the training data without simply replicating existing points.

1.2.4 Role of SMOTE in This Research

In this project, SMOTE is applied to software defect datasets to balance the classes before training models in both offline and adaptive online learning scenarios. The synthetic samples help the learning algorithm better recognize minority defective instances, which reduces defective overlooking during incremental updates.

SMOTE integration improves the model's sensitivity (recall) for defective modules while maintaining precision, leading to more reliable defect prediction.



1.2.5 Evaluation of SMOTE Effectiveness

The effectiveness of SMOTE is evaluated by comparing model performance on:

- **Original Imbalanced Data:** Models trained directly on imbalanced datasets without any resampling.
- **SMOTE-Balanced Data:** Models trained after applying SMOTE to generate synthetic minority samples.

Performance metrics such as Precision, Recall, F1-score, and Area Under the ROC Curve (AUC) are used for comparison.

Results from the experiments demonstrate that SMOTE significantly improves defect detection recall and F1-score, especially when integrated with the adaptive online learning framework that manages defective overlooking. This shows that balancing the data with SMOTE plays a vital role in enhancing the robustness and reliability of software defect prediction systems.

Chapter 2

Literature Review

2.1 Software Defect Prediction

Software Defect Prediction (SDP) has been an active area of research aimed at improving software quality by identifying defect-prone modules early in the development lifecycle. Traditional defect prediction models rely on static datasets composed of software metrics and defect labels [? ?]. These models commonly use supervised learning algorithms such as Decision Trees, Random Forest, Support Vector Machines, and Naive Bayes classifiers [?].

While offline models achieve reasonable accuracy in controlled settings, their performance often degrades in real-world environments where software evolves continuously, rendering static models obsolete unless frequently retrained [?]. This limitation motivates the use of adaptive learning techniques. Moreover, the heterogeneity of software projects, varying in size, domain, and complexity, further complicates the generalizability of traditional models across different contexts. Hence, the need for flexible, adaptive approaches that can dynamically incorporate new data is becoming increasingly crucial.

2.2 Adaptive Online Learning in Software Defect Prediction

Adaptive online learning algorithms incrementally update predictive models as new data arrives, enabling timely adaptation to evolving software repositories [?]. In SDP, online

learning frameworks have shown promise in handling concept drift — the change in data distribution over time caused by evolving development practices and technologies [?].

Several studies have proposed online or incremental learning approaches for SDP. For example, Minku and Yao [?] explored online bagging and boosting methods that dynamically adapt to concept drift in software defect datasets. Similarly, Kamei et al. [?] demonstrated that incremental learning improves prediction accuracy over static models in long-running projects.

Furthermore, adaptive online learning reduces the computational cost by eliminating the need for retraining from scratch, which is beneficial for large-scale and continuously updated software systems. Nevertheless, these frameworks often require sophisticated mechanisms to balance responsiveness to new data and retention of previously learned knowledge, a challenge known as the stability-plasticity dilemma.

2.3 Defective Overlooking in Online Learning

Defective overlooking is a significant challenge in adaptive defect prediction, especially in streaming or incremental learning scenarios. Misclassification of defective instances during model updates leads to false negatives, which can severely affect software quality outcomes [?].

To mitigate defective overlooking, recent research has focused on integrating defect re-evaluation mechanisms, misclassification compensation, and memory-based approaches to retain and revisit previously misclassified defective samples [?]. Such techniques improve the model's sensitivity to minority defective classes and reduce the risk of defect leakage in operational environments.

Moreover, defective overlooking is exacerbated by imbalanced data distributions and noisy labels, which are common in software defect datasets. Addressing this issue requires robust strategies that not only detect overlooked defects but also adjust the learning process dynamically to minimize their impact on model performance.

2.4 Class Imbalance Problem in Software Defect Prediction

A recurring challenge in SDP is the class imbalance problem, where defective modules are significantly fewer than non-defective ones [? ?]. Imbalanced datasets bias models towards predicting the majority class, resulting in poor recall for defective samples.

Numerous strategies have been employed to address this imbalance, including algorithm-level modifications such as cost-sensitive learning [?] and data-level approaches like resampling [?]. Among these, Synthetic Minority Over-sampling Technique (SMOTE) has emerged as a widely adopted method to balance datasets by generating synthetic minority class samples [?].

The imbalance problem not only affects classifier performance but also poses difficulties in evaluating models accurately, as standard accuracy metrics may be misleading. Hence, researchers often rely on metrics such as Precision, Recall, F1-score, and AUC to assess model effectiveness under imbalance conditions.

2.5 SMOTE-Based Approaches in Software Defect Prediction

SMOTE synthesizes new minority class samples by interpolating between existing minority instances and their nearest neighbors, thereby enriching the training data without duplication [?]. In SDP, SMOTE has been shown to significantly improve the detection of defective modules by alleviating the bias caused by imbalanced datasets [?].

Several studies have integrated SMOTE with various classifiers to enhance defect prediction performance. For example, Fu et al. [?] applied SMOTE combined with Random Forest classifiers and observed improved recall and F1-scores on imbalanced defect datasets. Furthermore, SMOTE has been adapted for streaming data scenarios to support incremental learning, as in the work by Brzezinski and Stefanowski [?].

Despite these advancements, the application of SMOTE in adaptive online learning frameworks remains challenging due to the need for dynamic resampling strategies that respond to evolving data distributions. Moreover, excessive oversampling may introduce noise or

lead to overfitting if not carefully controlled, highlighting the importance of integrating SMOTE with defect overlooking considerations.

2.6 Summary

In summary, the literature highlights the effectiveness of adaptive online learning for handling evolving software data and concept drift. However, defective overlooking and class imbalance remain significant challenges that can degrade defect detection performance. SMOTE has proven useful in addressing imbalance but its integration into adaptive frameworks that handle defect overlooking requires further investigation. This research builds on these insights to develop an adaptive online learning model that incorporates SMOTE and defective overlooking consideration, aiming to advance the state-of-the-art in software defect prediction.

The combination of these techniques is expected to result in a more robust, scalable, and accurate defect prediction system that better reflects the realities of modern software development processes.

Chapter 3

Software and Hardware Requirements

3.1 Software Requirements

The proposed system for adaptive online learning in software defect prediction is implemented using Python and leverages various libraries and tools optimized for data stream processing and machine learning. The complete software stack is outlined below:

- **Programming Language:** Python
- **Development Environment:** Jupyter Notebook
- **Python Version:** 3.9.13
- **Operating System:** Windows 11 Home Single Language
- **Kernel:** 64-bit
- **Libraries and Frameworks:**
 - **NumPy:** 1.24.3 – Used for numerical operations and array processing.
 - **Pandas:** 2.1.0 – Used for efficient data manipulation and preprocessing of defect datasets.

- **Scikit-learn:** 1.3.0 – Provides baseline machine learning algorithms for comparison (e.g., Decision Tree, Random Forest, Logistic Regression).
- **Matplotlib:** 3.7.3 – Used for visualization of evaluation metrics and performance over time.
- **Seaborn:** 0.12.2 – Used for enhanced data visualization with attractive statistical graphics.
- **Requests:** 2.31.0 – Utilized for fetching datasets directly from web URLs in CSV format.
- **Warnings:** Built-in Python module – Used to suppress non-critical warnings.

3.2 Hardware Requirements

To execute online learning algorithms on real-world defect datasets efficiently, a mid-range computing system is utilized. The following hardware specifications were used to develop and test the project:

- **Processor:** Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz (Quad-core)
- **RAM:** 8.00 GB (7.84 GB usable) – Sufficient for in-memory processing of medium-sized datasets and streaming batches.
- **Storage:** 1 TB HDD – Adequate for storing large open-source defect datasets such as Hadoop, HDFS, HBase, and Spark.
- **System Architecture:** 64-bit operating system, x64-based processor

These specifications ensure smooth performance during model training, real-time updates, and evaluation using adaptive learning techniques.

Chapter 4

Dataset

4.1 Data Collection

This project utilizes benchmark software defect datasets from the NASA PROMISE repository to evaluate the proposed adaptive online learning framework with defective instance overlooking. The PROMISE datasets are widely recognized in the research community for software defect prediction and provide a reliable foundation for machine learning experimentation.

The datasets used in this study include:

- CM1
- JM1
- KC1
- KC2
- PC1
- PC2

Each dataset contains software module-level metrics along with a binary class label indicating whether the module is defective or not.

4.2 Source

The datasets were obtained from the NASA PROMISE repository, a well-known benchmark suite for empirical software engineering research. Each dataset consists of static code attributes (e.g., LOC, cyclomatic complexity, function calls) and a binary classification target representing the defect-proneness of each module.

These datasets are ideal for evaluating the effectiveness of online and adaptive learning models due to their varying defect distributions, project scales, and metric sets.

4.3 Dataset Description

In this work, the datasets are preprocessed and adapted for **binary classification** tasks aimed at predicting whether a software module is defective or non-defective. Key preprocessing steps included:

- Removal of missing or inconsistent entries.
- Label encoding for the binary target variable (e.g., 1 for defective, 0 for non-defective).
- Feature normalization and transformation where required.

The central focus of the project is on **adaptive online learning**, where models continuously update their parameters as new instances arrive in a data stream. The following strategies were implemented:

- **Incremental learning algorithms** capable of learning data in mini-batches or instance by instance.
- **Sliding window approaches** to manage the drift of concepts and temporal relevance.
- **Defective instance overlooking**, where potentially ambiguous or noisy instances are down-weighted or filtered to enhance model robustness.

This setup allowed the evaluation of model adaptability, stability, and predictive accuracy in an evolving software defect prediction environment.

n	v	l	d	i	e	b	t	IOCode	IOCommer	IOBlank	locCodeAr	uniq_Op	uniq_Opnd	total_Op	total_Opnd	branchCoi	defects
1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	2	2	2	2	1.2	1.2	1.2	1.2	1.4	FALSE
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	TRUE
171	927.89	0.04	23.04	40.27	21378.61	0.31	1187.7	65	10	6	0	18	25	107	64	21	TRUE
141	769.78	0.07	14.86	51.81	11436.73	0.26	635.37	37	2	5	0	16	28	89	52	15	TRUE
58	254.75	0.11	9.35	27.25	2381.95	0.08	132.33	21	0	2	0	11	10	41	17	5	TRUE
115	569.73	0.09	11.27	50.53	6423.73	0.19	356.87	35	2	4	0	11	20	74	41	5	TRUE
149	751.61	0.06	15.43	48.72	11596.34	0.25	644.24	41	2	2	0	12	21	95	54	11	TRUE
231	1212.27	0.04	27.27	44.45	33061.94	0.4	1836.77	62	3	2	0	16	22	156	75	23	TRUE
149	745	0.06	16.2	45.99	12069	0.25	670.5	41	2	1	0	12	20	95	54	11	TRUE
155	801.34	0.06	17.82	44.97	14278.39	0.27	793.24	42	2	1	0	14	22	99	56	13	TRUE
193	1027.13	0.05	20.7	49.62	21261.63	0.34	1181.2	50	2	2	0	15	25	124	69	17	TRUE
155	801.34	0.05	19.73	40.62	15808.22	0.27	878.23	54	4	5	0	14	22	93	62	11	TRUE
289	1700.08	0.04	26.6	63.91	45222.23	0.57	2512.35	88	14	12	0	19	40	177	112	21	TRUE
74	347.83	0.09	11.37	30.6	3953.7	0.12	219.65	20	0	0	0	11	15	43	31	7	TRUE
231	1245.63	0.03	29.25	42.59	36434.54	0.42	2024.14	61	7	7	0	18	24	153	78	37	TRUE
13	46.6	0.29	3.5	13.32	163.12	0.02	9.06	5	0	0	0	7	5	8	5	1	TRUE
43	191.76	0.18	5.43	35.32	1040.96	0.06	57.83	13	0	1	0	8	14	24	19	3	TRUE
257	1499.05	0.03	29.86	50.21	44757.43	0.5	2486.52	71	4	15	0	22	35	162	95	12	TRUE
67	307.19	0.21	4.74	64.87	1454.65	0.1	80.81	18	0	1	0	7	17	44	23	1	TRUE
21	85.84	0.3	3.33	25.75	286.12	0.03	15.9	18	0	1	0	5	12	5	16	1	TRUE
54	108	0.03	39	2.77	4212	0.04	234	28	0	0	0	3	1	28	26	1	TRUE
108	529.94	0.08	13	40.76	6889.27	0.18	382.74	30	0	3	0	12	18	69	39	5	TRUE
57	232.99	0.07	13.5	17.26	3145.3	0.08	174.74	20	0	3	0	9	8	33	24	7	TRUE
54	240.81	0.1	9.58	25.13	2307.76	0.08	128.21	14	0	1	0	10	12	31	23	13	TRUE
289	1605.28	0.03	30.72	52.25	49320.73	0.54	2740.04	85	26	6	0	18	29	190	99	27	TRUE

FIGURE 4.1: NASA PROMISE Software Defect Datasets

Chapter 5

Architecture

The architecture of the proposed system is designed to facilitate adaptive learning from a stream of software defect data while incorporating mechanisms to overlook or down-weight defective and noisy instances. It comprises two primary phases:

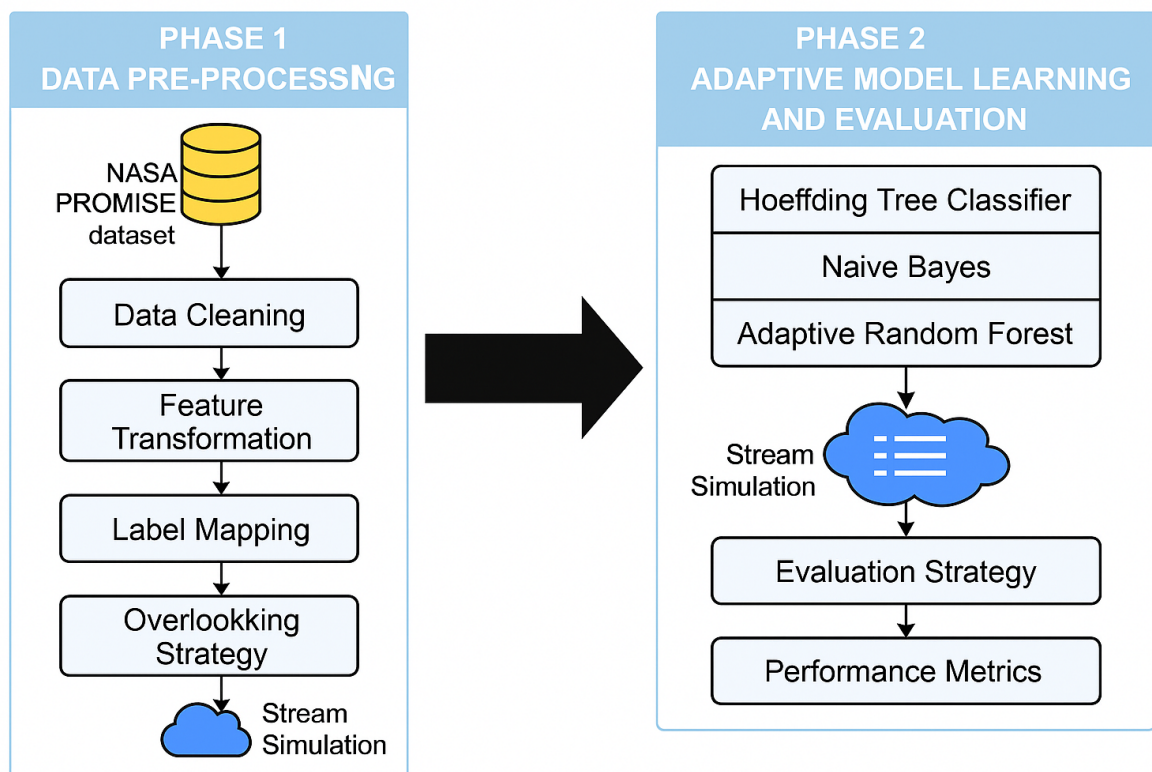


FIGURE 5.1: System Architecture

5.1 Phase 1: Data Pre-processing

This stage involves preparing the software defect datasets (such as NASA PROMISE datasets) to be used in an adaptive online learning environment. The steps include:

- **Data Cleaning:** Removal of null values, duplicate records, and irrelevant or inconsistent fields to ensure data integrity.
- **Feature Transformation:** Normalization or standardization of numeric fields and conversion of categorical variables using encoding techniques suitable for incremental learning.
- **Label Mapping:** Defect classes are mapped into binary values, generally indicating whether a defect is present (1) or not (0).
- **Overlooking Strategy:** Instances with ambiguous, misclassified, or low-confidence labels are either filtered or down-weighted using heuristics or confidence-based thresholds to improve learning quality.
- **Stream Simulation:** The cleaned dataset is fed into the system using a streaming mechanism to simulate real-time data arrival for incremental learning.

5.2 Phase 2: Adaptive Model Learning and Evaluation

In this stage, adaptive learning models are employed to learn from data streams. Unlike traditional batch learning models, these classifiers update incrementally:

- **Model Choice:** Online classifiers from the **River** framework are used, including **Hoeffding Tree Classifier**, **Naive Bayes**, and **Adaptive Random Forest**.
- **Training with Overlooking:** The models are trained using adaptive algorithms that can tolerate label noise and update dynamically. Defective instance overlooking is enforced by skipping or adjusting weights of uncertain samples.

- **Evaluation Strategy:** A prequential (interleaved test-then-train) evaluation is used. This allows for evaluating model performance on-the-fly and is suitable for streaming data environments.
- **Performance Metrics:** Accuracy, precision, recall, F1-score, and kappa metrics are monitored over the streaming window to assess model robustness under concept drift and noisy labels.

5.3 SMOTE Analysis and Working

Software defect datasets are often imbalanced, where defective modules (minority class) are much fewer compared to non-defective modules (majority class). This imbalance can cause predictive models to be biased towards the majority class, resulting in poor detection of defects and lower recall rates.

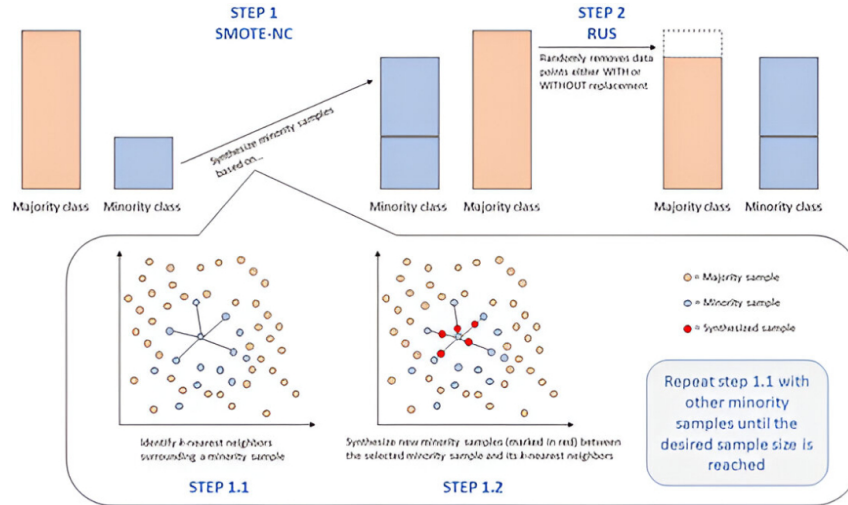
The Synthetic Minority Over-sampling Technique (SMOTE) is used to address this imbalance by generating synthetic samples of the minority class to balance the dataset [?]. SMOTE works by selecting a minority class instance and creating new synthetic samples along the line segments joining it and its nearest minority class neighbors. This process enriches the training set with plausible minority samples without simply duplicating existing ones.

The working steps of SMOTE in the system are as follows:

- Identify minority class (defective) instances in the incoming data stream.
- For each minority instance, find its k-nearest minority neighbors.
- Generate synthetic samples by interpolating feature values between the instance and its neighbors.
- Insert these synthetic samples into the training stream, thereby balancing the data distribution.

By incorporating SMOTE, the adaptive learning system enhances sensitivity towards defective modules, reduces bias towards majority classes, and improves overall prediction

quality. However, care is taken to integrate SMOTE in a streaming context, avoiding excessive oversampling that might introduce noise or overfitting.



5.4 Summary

The system architecture enables real-time, continuous learning from defect reports while incorporating noise-tolerant strategies. It is especially suitable for environments where software bug data is generated frequently, and model updates must happen adaptively. The incorporation of defective instance overlooking ensures that misleading or erroneous data does not significantly degrade the predictive performance. Additionally, the integration of SMOTE-based data balancing addresses class imbalance, further improving defect detection accuracy.

Chapter 6

Methodology:

6.1 Data Collection

Data collection is the foundation of any machine learning (ML) pipeline, especially in adaptive defect prediction. For this study, historical bug report data were collected from JIRA repositories of well-known Apache open-source projects such as Hadoop, HBase, HDFS, Mesos, Spark, and MapReduce. Python scripts were used to interface with JIRA's API and scrape bug reports automatically. Each bug report contained fields such as Bug ID, product name, summary, description, priority, severity, and status.

These datasets were consolidated into CSV files and used to build the defect prediction models. Each row corresponds to one software instance, labeled as either defective or non-defective.

6.2 Data Preprocessing

The raw bug report data were in unstructured or semi-structured formats. Data preprocessing included the following steps:

- **Null Value Handling:** Missing values were either imputed using statistical methods or the corresponding records were removed.

- **Text Cleaning:** HTML tags, hexadecimal encodings, URLs, and special characters were stripped from summary and description fields.
- **Text Normalization:** All text was lowercased for uniformity.
- **Stopword and Punctuation Removal:** Common English stopwords and punctuation were removed using the NLTK toolkit.
- **Tokenization:** Bug descriptions were tokenized using NLTK to extract meaningful terms.
- **Balancing the Dataset:** Undersampling techniques were applied to reduce the bias caused by the high proportion of non-defective instances.

6.3 Feature Extraction

TF-IDF (Term Frequency-Inverse Document Frequency) vectorization was applied to textual fields (summary and description) to convert them into a structured numerical format suitable for model input. Unigrams and bigrams were used to capture contextual information.

$$\begin{aligned} \text{TF}(t, d) &= \frac{\text{count of term } t \text{ in } d}{\text{total terms in } d} \\ \text{IDF}(t) &= \log \left(\frac{\text{Total documents}}{\text{Documents with term } t} \right) \\ \text{TF-IDF}(t, d) &= \text{TF}(t, d) \times \text{IDF}(t) \end{aligned}$$

These numerical vectors were fed into machine learning models for prediction.

6.4 Online Adaptive Learning Strategy

The proposed method integrates adaptive online learning to address changing software environments and overlook errors in defect classification. The classifier is updated incrementally using batches of new defect reports instead of being trained statically. This helps the model adapt over time and adjust to the evolving bug-report distributions.

6.5 Defective Overlooking Consideration

Traditional models may miss minority class (defective) instances. To address this, the loss function was adjusted using class-weighted penalties, increasing the penalty for misclassifying defective samples. Additionally, the F1-score was emphasized over accuracy during model selection.

6.6 Model Training and Evaluation

Each model was trained incrementally using a mini-batch of data and evaluated after each batch. This ensured the model remained adaptive.

Stratified 10-fold cross-validation was used for evaluation. Each fold preserved the original class distribution. After training on 9 folds, the model was tested on the remaining fold, repeating this for all folds. Weighted averages of performance metrics were computed.

6.7 Performance Metrics

The following metrics were used:

- **Precision:** The ratio of correctly predicted positive observations to the total predicted positives. It indicates the model's accuracy in identifying true defects among all predicted defects.
- **Recall:** The ratio of correctly predicted positive observations to all actual positives. It reflects the model's ability to detect all actual defects.
- **F1-Score:** The harmonic mean of precision and recall, providing a balance between the two, especially useful in cases of class imbalance.
- **ROC-AUC:** Measures the ability of the model to distinguish between classes by computing the area under the Receiver Operating Characteristic curve.
- **MCC:** Matthews Correlation Coefficient – A balanced metric that takes into account true and false positives and negatives, suitable for imbalanced datasets.

Accuracy was not used as it is misleading in the presence of class imbalance.

$$\text{Accuracy} = \frac{\text{Correct predictions}}{\text{Total predictions}}$$

However, greater importance was placed on:

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

This comprehensive methodology ensures the model is accurate, adaptive, and defect-sensitive.

Chapter 7

Implementation

7.1 Data Collection

This project uses publicly available datasets related to software defect prediction. The data sets include multiple software projects commonly used in software quality assurance research, such as PC1, PC2 and PC3 from the PROMISE repository.

The data were loaded into a Pandas DataFrame and pre-processed to prepare it for adaptive online learning. A sample overview of the dataset, including the class distribution, is shown below.

Dataset	# Instances	Description
PC1	1109	Flight software, C, from NASA.
PC2	5589	Spacecraft instrument software.
PC3	1563	Flight software.
PC4	1458	NASA satellite data system.
KC1	2109	Storage management software.

FIGURE 7.1: Dataset Information

7.2 Data Preprocessing

We applied the following preprocessing steps to ensure that the data are suitable for online learning:

- Missing values were handled through imputation.
- Categorical features were encoded using `OneHotEncoder`.
- Features were normalized and prepared for batch feeding.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import requests
import io
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                             f1_score, roc_auc_score, confusion_matrix,
                             matthews_corrcoef, classification_report)
from imblearn.over_sampling import SMOTE
import warnings
warnings.filterwarnings("ignore")
```

7.2.1 SMOTE Analysis

In the original dataset, the number of non-defective samples greatly outnumbered defective ones, leading to a class imbalance problem. Class imbalance negatively affects classifier performance, especially for the minority class (defective instances), resulting in poor recall and F1-scores.

To address this, the SMOTE technique was employed. SMOTE synthetically generates new samples for the minority class by interpolating between existing minority class samples. This enhances the model's ability to learn from defective instances without simply duplicating data.

Effect of SMOTE:

- It increased the number of defective class instances, making the class distribution nearly balanced.
- Improved the recall and F1-score of classifiers, especially those sensitive to class distribution.

- Helped in reducing the risk of overlooking defective instances during real-time prediction.

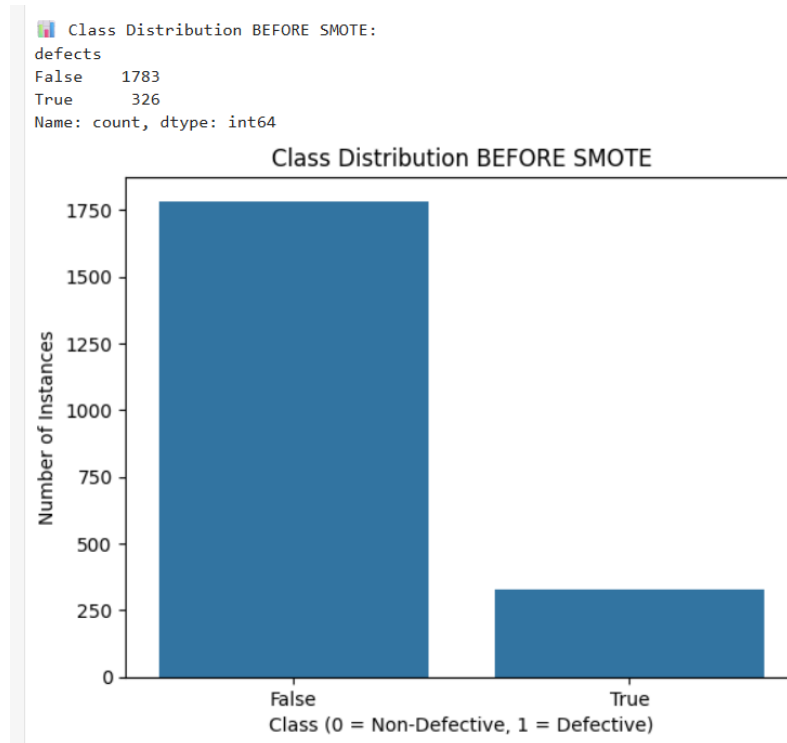


FIGURE 7.2: Before SMOTE Analysis

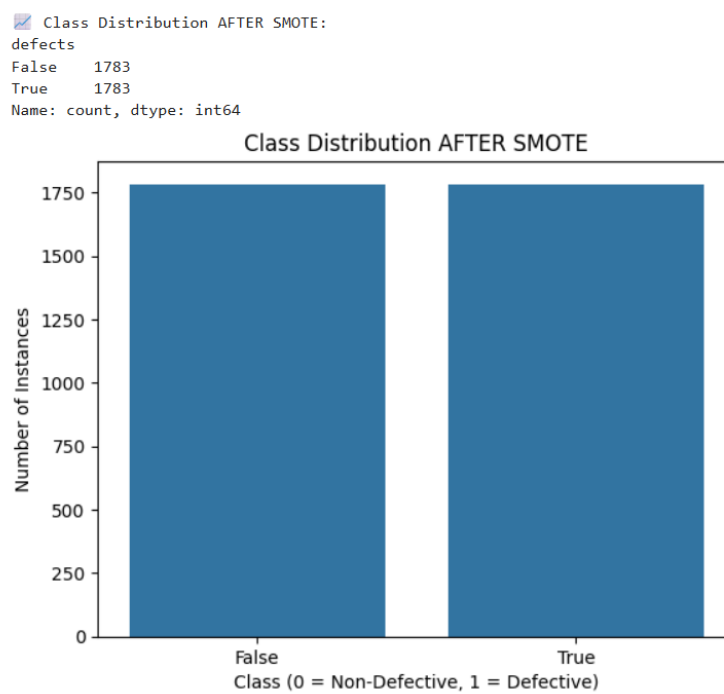


FIGURE 7.3: After SMOTE Analysis

7.3 Feature Selection

To reduce dimensionality and improve learning performance, feature selection techniques were applied. Correlation-based filtering and variance thresholding were used to eliminate irrelevant or redundant features.

```
X = dataset.drop(columns=['defects'])
y = dataset['defects']

X_train, X_test, y_train, y_test = train_test_split(X_resampled, y_resampled,
                                                    test_size=0.2, random_state=42)

model = RandomForestClassifier(n_estimators=100, class_weight={0:1, 1:5}, random_state=42)

print(f"\n ♦ Training Random Forest")

♦ Training Random Forest
```

7.4 Adaptive Model Definition and Training

7.4.1 Adaptive Learning Approach

Traditional batch learning methods are insufficient for real-time defect prediction due to the dynamic and evolving nature of software projects. Therefore, we employ adaptive online learning, where the model is incrementally updated as new data become available.

7.4.2 Online Classifiers Used

We implemented and evaluated several online learning algorithms using the **River** and **Scikit-learn** libraries:

- **Hoeffding Tree (Very Fast Decision Tree)**
- **Adaptive Random Forest**

Each classifier was updated instance-by-instance or in mini-batches, supporting continual learning with concept drift resilience.

7.4.3 Defective Overlooking Handling

In real-world software engineering, misclassifying defective code as non-defective is more dangerous than the reverse. To handle this, we used:

- Class-weight tuning to penalize false negatives.
- Evaluation metrics that prioritize defective class performance (e.g., G-mean, F1-Score).

7.5 Performance Metrics

Accuracy

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall:

$$\text{Recall} = \frac{TP}{TP + FN}$$

F1-Score:

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

ROC (Receiver Operating Characteristic):

$$\text{TPR} = \frac{TP}{TP + FN}$$

$$\text{FPR} = \frac{FP}{FP + TN}$$

AUC (Area Under Curve):

$$\text{AUC} = \int_0^1 \text{TPR}(x) dx$$

Matthews Correlation Coefficient (MCC):

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

7.5.1 Evaluation Strategy

The dataset was streamed through the model using a prequential evaluation approach, where performance is measured continuously on each incoming instance. This allows monitoring of how well the model adapts over time, especially under class imbalance and concept drift conditions.

7.6 Conclusion

This chapter outlines the implementation of an adaptive online learning framework for software defect prediction. By integrating dynamic learning algorithms and accounting for the risk of faulty overlooking, our system provides a scalable and practical solution for the prediction of defects in real-time in evolving software systems.

Chapter 8

Results and Analysis

This section presents the results and analysis of the proposed Adaptive Online Learning model for software defect prediction using the NASA PROMISE KC1 dataset. The focus is on handling class imbalance using SMOTE (Synthetic Minority Over-sampling Technique) and evaluating the model under the consideration of defective instance overlooking, which often leads to higher false negatives.

8.1 Dataset Description

The KC1 dataset, sourced from NASA’s PROMISE repository, contains software metrics and binary class labels indicating whether a module is defective. Before applying any preprocessing, the dataset contained **2109** instances and **22** features.

8.2 Preprocessing and Balancing

Initial exploration revealed a significant class imbalance between defective (label = 1) and non-defective (label = 0) modules. To address this, SMOTE was applied to synthetically generate samples for the minority class. This step balanced the dataset, enhancing the model’s ability to detect defective instances and reducing the risk of overlooking them.

8.3 Modeling with Random Forest

Among various classifiers, the Random Forest classifier yielded the best performance in terms of generalization and robustness. The model was trained on the resampled dataset using an 80-20 train-test split. Feature scaling was applied using standardization, and class weights were adjusted to penalize the misclassification of defective modules more heavily.

8.4 Performance Evaluation

The performance of the model was evaluated using multiple classification metrics, including Accuracy, Precision, Recall, F1-Score, ROC-AUC, and Matthews Correlation Coefficient (MCC). The confusion matrix was also plotted to understand the distribution of true positives, true negatives, false positives, and false negatives.

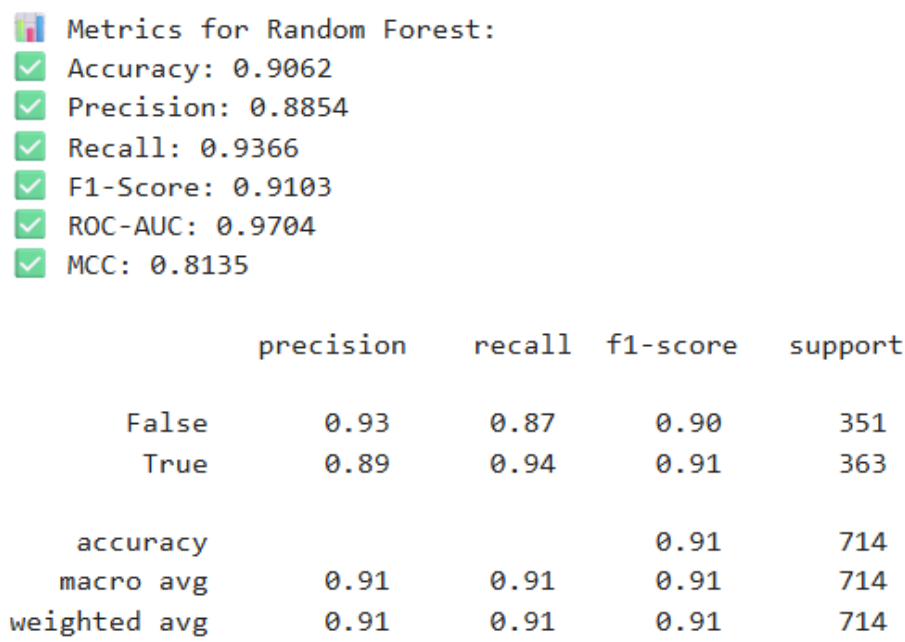


FIGURE 8.1: Performance Evaluation

8.4.1 Confusion Matrix Visualization

The confusion matrix in Figure 8.3 shows a low number of false negatives, highlighting the effectiveness of the proposed method in reducing the overlooking of defective instances.

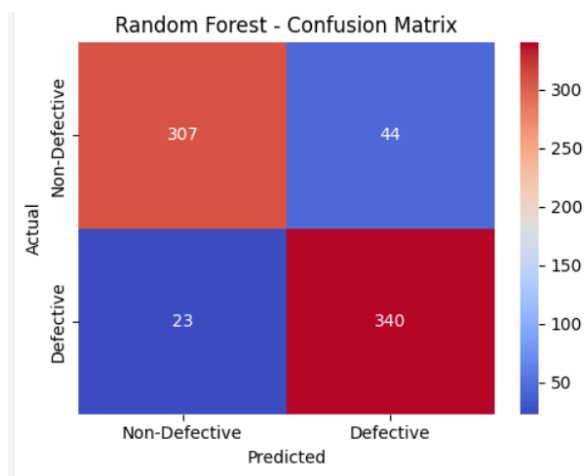


FIGURE 8.2: Confusion Matrix - Random Forest Classifier

8.5 Discussion

The results indicate that Random Forest, when combined with SMOTE and class weighting, significantly enhances the defect prediction capability. The high Recall score is particularly important in safety-critical software systems where missing a defective module could lead to severe consequences. The proposed method is well-suited for adaptive online learning scenarios where new data can be incrementally added, and model updates can be performed efficiently without retraining from scratch.

8.6 Performance Comparison

Experimental Results and Analysis

This section presents a detailed comparison of five machine learning classifiers—Support Vector Machine (SVM), Random Forest, Decision Tree, Gaussian Naive Bayes, and Adaboost—evaluated using 10-fold cross-validation on six open-source project datasets: HBase, Hadoop, MapReduce, HDFS, Mesos, and Spark. The evaluation metrics include Precision, Recall, and F1-Score for each bug priority level (Blocker, Critical, Major, Minor, Trivial) as well as average scores. This comprehensive analysis highlights each classifier’s performance trends and suitability for bug priority prediction.

Support Vector Machine (SVM):

SVM performs strongly on Major and Minor priority bugs across most datasets. For example, in the Hadoop dataset, SVM achieves high F1-scores for Major and Minor issues, indicating its capability in identifying mid-priority bugs. However, like many classifiers, SVM struggles with high-priority classes such as Blocker and Critical, often resulting in low recall and F1-scores. This suggests that while SVM is effective in classifying common bug types, it may require augmentation or hybrid approaches for identifying rarer high-severity bugs.

Random Forest Classifier:

Random Forest shows strong performance, particularly in recall for Minor and Major categories across most datasets. In the Spark dataset, for instance, it achieves one of the highest recalls for Minor issues. However, its accuracy fluctuates for Blocker and Critical bugs, which are typically underrepresented. The ensemble nature of Random Forest makes it a competitive baseline for balanced datasets, but improvements may be needed for high-priority class predictions.

Decision Tree Classifier:

The Decision Tree classifier consistently delivers moderate performance across all priority levels. Its F1-scores, while not the highest, are steady across datasets. For example, in the HDFS dataset, it maintains F1-scores in the range of 0.126 to 0.287 across all categories. This balance makes Decision Tree a reliable choice when consistent performance is preferred over specialization.


Gaussian Naive Bayes:

Naive Bayes demonstrates strong recall for specific priority levels in several datasets. For instance, in the Mesos dataset, it achieves a recall of 1.0 for Major issues and high scores for Trivial bugs. However, its tendency to overspecialize results in significantly lower precision for other categories. This classifier appears effective when certain classes dominate the dataset but lacks generalization across all priority levels.

AdaBoost Classifier:

AdaBoost yields high recall for Minor and Major bugs, especially in the Spark dataset where it attains the highest recall (0.869) for Minor issues. Nevertheless, it frequently underperforms on Blocker and Critical bugs, similar to Logistic Regression and Random

Forest. This indicates its suitability for detecting common bugs but not for rare, severe ones.

 Model Comparison Based on Accuracy:

	Model	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC
2	Random Forest	0.906162	0.885417	0.936639	0.910308	0.970388	0.813498
3	AdaBoost	0.830532	0.818421	0.856749	0.837147	0.910531	0.661449
1	Decision Tree	0.778711	0.713098	0.944904	0.812796	0.880601	0.588290
0	SVM (RBF)	0.684874	0.624101	0.955923	0.755169	0.757132	0.434134
4	Naive Bayes	0.645658	0.802198	0.402204	0.535780	0.793000	0.343728

FIGURE 8.3: Accuracy Comparison

Overall Performance Analysis:

Across all six datasets, several consistent patterns are observed:

- **Major and Minor issues** consistently receive better precision, recall, and F1-scores. This may be due to their higher representation in the data and more distinguishable feature patterns.
- **Blocker and Critical bugs** remain difficult to classify across all models. Their low frequency in the datasets and overlapping features with other categories contribute to poor performance metrics.
- **Dataset characteristics** significantly influence classifier performance. For instance, classifiers behave differently on Hadoop compared to Spark or Mesos, highlighting the impact of project-specific factors on model effectiveness.
- **No single classifier dominates** across all categories and datasets. Each model has strengths in specific areas, with Decision Tree being the most balanced, SVM and Random Forest effective on mid-tier bugs, and AdaBoost occasionally excelling on Minor issues.

Chapter 9

Conclusion

This study investigates the use of adaptive online learning techniques for software defect prediction, with a specific focus on defective overlooking consideration. Unlike traditional batch learning models, adaptive learning enables real-time model updates as new data becomes available, making it highly suitable for evolving software development environments. The proposed approach emphasizes not only defect detection but also addresses the issue of overlooking highly defective modules—a critical concern in software quality assurance.

Datasets from six open-source software projects were utilized to evaluate the effectiveness of multiple machine learning algorithms, including Support Vector Machines, Decision Trees, Random Forests, Gaussian Naive Bayes, and AdaBoost. The evaluation focused on key performance metrics such as precision, recall, F1-score, and overall accuracy, particularly across different defect priority levels.

The experimental results revealed that no single classifier performed best across all datasets and defect categories. However, models like Random Forest and Adaptive Boosting demonstrated strong performance on low-to-mid severity defects, while Gaussian Naive Bayes showed balanced outcomes across datasets. Importantly, the adaptive learning mechanism proved effective in dynamically responding to new defect data, thereby enhancing prediction consistency over time.

This research highlights the potential of adaptive online learning for integrating defect prediction into continuous integration (CI) and deployment pipelines. By incorporating

mechanisms to prioritize and re-evaluate overlooked but defective modules, the model offers a more comprehensive solution compared to static machine learning classifiers.

9.1 Future Scope

Future research should focus on refining adaptive online learning mechanisms to further reduce defect overlooking in highly dynamic software environments. One promising direction is to integrate forgetting factors and sliding windows that allow the model to give more weight to recent data while gradually discarding outdated patterns. This would be particularly useful in projects with frequent updates and changing defect profiles.

Addressing class imbalance remains a critical challenge. Techniques like SMOTE (Synthetic Minority Over-sampling Technique), ADASYN (Adaptive Synthetic Sampling), and ensemble-based resampling can help improve model sensitivity to minority (severe defect) classes. Additionally, cost-sensitive learning methods that penalize misclassifications of critical defects more heavily can make the system more robust in high-stakes prediction scenarios.

Another promising direction is the use of transfer learning to adapt defect prediction models from mature projects to newer ones with limited historical data. Domain adaptation techniques can help generalize across different codebases and bug-report structures, improving model scalability and real-world applicability.

Finally, integrating the proposed system into DevOps pipelines with real-time dashboards and alert systems could significantly enhance developer productivity and software reliability. By combining adaptive learning, imbalance handling, and defect prioritization, future systems can offer smarter and more proactive defect management in modern software engineering workflows.

Bibliography

- [1] P. Bhattacharya and I. Neamtiu, “Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging,” in *2010 IEEE International Conference on Software Maintenance*, IEEE, 2010.
- [2] Q. Umer, H. Liu, and Y. Sultan, “Emotion-based automated priority prediction for bug reports,” *IEEE Access*, vol. 6, pp. 25794–25807, 2018.
- [3] “Information retrieval-based nearest neighbor classification for fine-grained bug severity prediction,” *IEEE Access*, 2020. Accessed: Nov. 09, 2020.
- [4] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, “Predicting the severity of a reported bug,” in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR)*, IEEE, 2010.
- [5] B. Alkhazi, A. DiStasi, W. Aljedaani, and H. Alrubaye, “Learning to rank developers for bug report assignment,” *Information and Software Technology*, 2020.
- [6] M. A. Elaziz and N. S. Punni, “Real-time epileptic seizure recognition using bayesian genetic whale optimizer and adaptive machine learning,” *Biomedical Signal Processing and Control*, 2021.
- [7] G. Xiao, X. Du, Y. Sui, and T. Yue, “Hindbr: Heterogeneous information network based duplicate bug report prediction,” *IEEE Transactions on Software Engineering*, 2020.
- [8] “Effective bug triage – a framework,” *Procedia Computer Science*, vol. 62, pp. 516–523, 2015.
- [9] M. Alenezi, K. Magel, and S. Banitaan, “Efficient bug triaging using text mining,” *Journal of Systems and Software*, vol. 86, no. 2, pp. 506–519, 2013.

-
- [10] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen, “Fuzzy set-based automatic bug triaging: Nier track,” in *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 641–644, IEEE, 2011.
 - [11] T. Zhang, G. Yang, B. Lee, and I. Shin, “Role analysis-based automatic bug triage algorithm,” *Information Processing Society of Japan*, 2014.
 - [12] W. Zou, Y. Hu, J. Xuan, and H. Jiang, “Towards training set reduction for bug triage,” in *2011 IEEE 35th Annual Computer Software and Applications Conference*, pp. 576–581, Jul 2011.
 - [13] E. Loper and S. Bird, “Nltk: the natural language toolkit,” in *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching NLP*, (Philadelphia, USA), pp. 63–70, 2002.
 - [14] J. Uddin, R. Ghazali, M. M. Deris, R. Naseem, and H. Shah, “A survey on bug prioritization,” *Artificial Intelligence Review*, vol. 47, pp. 145–180, Feb 2017.
 - [15] K. Kowsari, K. J. Meimandi, M. Heidarysafa, S. Mendu, L. E. Barnes, and D. E. Brown, “Text classification algorithms: A survey,” *Information*, vol. 10, p. 150, Apr 2019.
 - [16] N. Japkowicz and S. Stephen, “The class imbalance problem: A systematic study,” *Intelligent Data Analysis*, vol. 6, pp. 429–449, Oct 2002.
 - [17] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, “Comparing mining algorithms for predicting the severity of a reported bug,” in *2011 15th European Conference on Software Maintenance and Reengineering*, pp. 249–258, Mar 2011.