FORM NO. - F/TL/024

Rev.00. Date 20.03.2020

# A NOVEL MACHINE LEARNING METHOD FOR SOFTWARE DEFECT ESTIMATION

PROJECT REPORT-Phase II

*submitted in partial fulfillment of the requirements*

*for the award of the degree in*

**BACHELOR OF TECHNOLOGY**

**in**

**COMPUTER SCIENCE AND ENGINEERING**

by

|  |  |
| --- | --- |
| **MANINDRA.G** | **(211061101143)** |
| **VINAY.G** | **(211061101145)** |
| **RAGHURAM REDDY.G** | **(211061101124)** |



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**APRIL 2025**

FORM NO. - F/TL/024

Rev.00. Date 20.03.2020

****

**DECLARATION**

We **MANINDRA.G (211061101143)**, **VINAY.G (211061101145)**, **RAGHURAM REDDY.G (211191101160)**, hereby declare that the (Project Phase-I) entitled “**A NOVEL MACHINE LEARNING METHOD FOR SOFTWARE DEFECT ESTIMATION** ” is done by us under the guidance of **Mr. M. ARUN** and is submitted in partial fulfillment of the requirements for the award of the degree in **BACHELOR OF TECHNOLOGY** in **Computer Science And Engineering** specialization in **Data Science(Artificial Intelligence)**.

**DATE:**

**PLACE:** CHENNAI

1.

2.

3.

**SIGNATURE OF THE CANDIDATE(S)**

FORM NO. - F/TL/024

Rev.00. Da

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**BONAFIDE CERTIFICATE**

This is to certify that this Project Report (Project Phase-I) is the bonafide work of Mr. **MANINDRA.G** Reg. No **211061101143,** Mr.  **VINAY.G** Reg.No**211061101145**, Mr. **RAGHURAM REDDY.G** Reg. No **211061101124**, who carried out the project entitled  **“A NOVEL MACHINE LEARNING METHOD FOR SOFTWARE DEFECT ESTIMATION** ” under our supervision from December 2024 to April 2025.

.

|  |  |  |
| --- | --- | --- |
| **Internal Guide**  **Mr. M. ARUN**  **Assistant Professor**  **Department of CSE**  **Dr. MGR Educational and Research Institute, Deemed to be University** | **Project Coordinator Dr M.Chandran**  **Professor**  **Department of CSE**  **Dr. MGR Educational and Research**  **Institute, Deemed to be University** | **Department Head Dr. S. GEETHA**  **Professor & HOD (CSE)**  **Department of CSE**  **Dr. MGR Educational and Research Institute, Deemed to be University** |

## Submitted for Viva Voce Examination held on

**Internal Examiner External Examiner**

(Name in Capital letters (Name in Capital letters

with Signature) with Signature)

FORM NO. - F/TL/024

Rev.00. Date 20.03.2020

## ACKNOWLEDGEMENT

We would first like to thank our respected Chancellor **Thiru. Dr. A.C.S. SHANMUGAM, B.A., B.L.,** President Er. A.C.S. ARUNKUMAR,**B.Tech., M.B.A.,** and Secretary **Thiru A.RAVIKUMAR** for all the encouragement and support extended to us during the tenure of this project and also our years of studies in his wonderful University.

We express my heartfelt thanks to our Vice Chancellor **Prof. Dr. S. GEETHALAKSHMI** in providing all the support of our Project (Phase - I).

We express my heartfelt thanks to our Head of the Department ,**Prof. Dr. S.GEETHA**, who has been actively involved and very influential from the start till the completion of our project.

Our sincere thanks to our Project Coordinators **Prof. Dr .G. Victo Sudha George,**

**Prof. Dr M. Chandran** and Project guide **Mr. M. ARUN** for their continuous guidance and encouragement throughout this work, which has made the project a success .

We would also like to thank all the teaching and non teaching staffs of Computer Science and Engineering department, for their constant support and the encouragement given to us while we went about to achieving our project goals.

# CONTENT

**CHAPTER NO. TITLE PAGE NO. LIST OF TABLES** i

**LIST OF FIGURES** ii

[**LIST OF ABBREVIATIONS** iii](#_bookmark0)

[**LIST OF KEYWORDS** iv](#_bookmark1)

[**ABSTRACT** v](#_bookmark2)

**MAJOR DESIGN CONSTRAINTS AND DESIGN** vi

1. **INTRODUCTION 1**
2. **REQUIREMENT ANALYSIS 13**
   1. [Literature Survey 13](#_TOC_250038)
   2. [Research Methodology 23](#_TOC_250037)
   3. Existing System 25
   4. Proposed System 26
   5. System Architecture

**3 REQUIREMENT SPECIFICATION**

**3.1 Functional Requirements** 27

3.1.1.Usability 27

3.1.2.Robustness 27

3.1.3.Security 27

3.1.4.Reliability 27

3.1.5.Flexibility 27

3.1.6.Safety 27

**3.2.Non-Functional Requirements** 28

3.2.1.Portability 28

3.2.2.Performance 28

3.2.3.Accuracy 28

3.2.4.Maintainability 28

**3.3.HARDWARE REQUIREMENTS** 29

**3.4.SOFTWARE REQUIREMENTS** 29

**4 DESIGN** 30

4.1 Work Flow Diagram 31

4.2 Data Flow 32

**4.3** **UML Diagrams**

4.3.1 Use Case Diagram 33

4.3.2 Class Diagram 34

4.3.3 Activity Diagram 35

4.3.4 Sequence Diagram 36

5 IMPLEMENTATION 38

**MODULES:** 38

5.1.Data Collection 39

5.2.Data Preparation 40

5.3.Model Selection 41

5.4.Analyse and Prediction 42

5.5.Code

**6 RESULTS AND DISCUSSION** 62

6**.**1 GRAPHS 62

6.2 OUTPUTS 65

**7 CONCLUSION**  43

**REFERENCE** 45

# LIST OF TABLES

|  |  |  |
| --- | --- | --- |
| **TABLE No.** | **TABLE NAME** | **PAGE No.** |

2.1.1 Literature Survey 13

5.2 Data Collection 39

**LIST OF FIGURES**

|  |  |  |
| --- | --- | --- |
| **FIGURE No** | **FIGURE NAME** | **PAGE NO** |
| 1 | General architecture | 13 |
| 2.3 | Existing system architecture | 28 |
| 2.4 | Proposed system Architecture | 31 |
| 4.1.1 | Workflow Diagram | 36 |
| 4.2.1 | Data Flow Diagram | 38 |
| 4.3.1.1 | Use Case Diagram | 40 |
| 4.3.2.1 | Class Diagram | 41 |
| 4.3.3.1 | Activity Diagram | 42 |
| 4.3.4.1 | Sequence Diagram | 43 |
| 5.1 | Data Collection | 45 |
| 5.2 | Data Preparation | 46 |
| 5.3 | Model Selection | 47 |

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |

# LIST OF ABBREVIATIONS

1. **SVM** - Support Vector Machine
2. **NB** - Naïve Bayes
3. **DT -** Decision Tree
4. **RNNS -** Recurrent Neural Networks
5. **SMOTE** - Synthetic Minority Over-sampling Technique
6. **XAL** - Explainable AI
7. **LOC** - Lines of Code
8. **DIT** - Depth of Inheritance Tree
9. **SMOTE -** Synthetic Minority Oversampling Technique
10. **CBO** - Coupling Between Objects
11. **LCOM** - Lack of Cohesion of Methods
12. **SQA** - software quality assurance

# LIST OF KEYWORDS

**S.No. Keywords**

* 1. Software Defect Prediction
  2. Software Quality Improvement
  3. Machine Learning
  4. Software Metrics
  5. Data Cleaning
  6. Prediction Accuracy
  7. SVM (Support Vector Machine),
  8. Naïve Bayes
  9. Decision Tree
  10. Performance Metrics
  11. Accuracy Rate
  12. Error Rate
  13. Dataset Optimization
  14. Software Testing Efficiency
  15. Random Forest

# ABSTRACT

Software defect prediction plays an important role in improving software quality and it help to reducing time and cost for software testing. Machine learning focuses on the development of computer programs that can teach themselves to grow and change when exposed to new data. The ability of a machine to improve its performance based on previous results. Machine learning improves efficiency of human learning, discover new things or structure that is unknown to humans and find important information in a document. For that purpose, different machine learning techniques are used to remove the unnecessary, erroneous data from the dataset. Software defect prediction is seen as a highly important ability when planning a software project and much greater effort is needed to solve this complex problem using a software metrics and defect dataset. Metrics are the relationship between the numerical value and it applied on the software therefore it is used for predicting defect. The primary goal of this survey paper is to understand the existing techniques for predicting software defect .Results obtained states that the proposed approach works more efficiently in terms of accuracy as compared to other techniques like SVM, Naïve Bayes, and Decision Tree. The results obtained by proposed method are having the values for performance time (min) is 3.24 minutes, Accuracy rate (%) is 96.78 %, and the Error rate (%) is 0.21 %.

**KEYWORDS:**

Software Defect Prediction, Machine Learning, Software Quality Improvement, SoftwareMetrics, Data Cleaning, Prediction Accuracy, SVM (Support Vector Machine), Naïve Bayes, Decision Tree, Performance Metrics, Accuracy Rate, Error Rate, Dataset Optimization, Software Testing Efficiency

MAJOR DESIGN CONSTRAINTS AND DESIGN STANDARDS TABLE

|  |  |  |  |
| --- | --- | --- | --- |
| Student Group | G.MANINDRA  (211061101143) | G.VINAY  (211061101145) | G.RAGHURAM  (211061101124) |
| Project Title | A NOVEL MACHINE LEARNINGMETHOD FOR SOFTWARE DEFECT ESTIMATION | | |
| Program Concentration  Area | SOFTWARE DEFECT(ERRORS) | | |
| Constraints Example | Power Constraints | | |
| Economic | Yes | | |
| Environmental | Yes | | |
| Sustainability | Yes | | |
| Implementable | Yes | | |
| Ethical | N/A | | |
| Health and Safety | Yes | | |
| Social | Yes | | |
| Political | No | | |
| Other | Power Modulation from Solar Panel | | |
| Standards |  | | |
| 1 | UART, SPI ,I2C | | |
| 2 | ISO 50001, ISO 12100 | | |
| 3 | USB to TTL | | |
| Prerequisite Courses for the Major Design  Experiences | 1. Microcontrollers 2. Embedded System 3. Electrical Circuits and Electronics | | |

# CHAPTER 1

INTRODUCTION

### 1.1 Background and Overview of software defect estimation

### 1.1.1 Introduction to software defect estimation

### Software defect prediction plays an important role in improving software quality and it help to reducing time and cost for software testing. Machine learning focuses on the development of computer programs that can teach themselves to grow and change when exposed to new data. The ability of a machine to improve its performance based on previous results. Machine learning improves efficiency of human learning, discover new things or structure that is unknown to humans and find important information in a document. For that purpose, different machine learning techniques are used to remove the unnecessary, erroneous data from the dataset. Software defect prediction is seen as a highly important ability when planning a software project and much greater effort is needed to solve this complex problem using a software metrics and defect dataset.

### Metrics are the relationship between the numerical value and it applied on the software therefore it is used for predicting defect. The primary goal of this survey paper is to understand the existing techniques for predicting software defect .Results obtained states that the proposed approach works more efficiently in terms of accuracy as compared to other techniques like SVM, Naïve Bayes, and Decision Tree. The results obtained by proposed method are having the values for performance time (min) is 3.24 minutes, Accuracy rate (%) is 96.78 %, and the Error rate (%) is 0.21 %.

### Machine learning (ML) has proven highly effective in software defect prediction due to its ability to handle large datasets and learn from historical data. ML models in this domain analyze past instances of software metrics and defect occurrences, learning patterns that correlate certain metrics with defect-prone areas. When these models are trained effectively, they can generalize and identify similar patterns in new data, predicting which parts of the software may need additional attention.

### Commonly used techniques in software defect prediction include Support Vector Machines (SVM), Naïve Bayes, Decision Trees, and more advanced methods such as Random Forest, Gradient Boosting, and Deep Learning models. Each technique brings distinct strengths:

Software defects, commonly referred to as bugs or failures, occur when a software product does not meet specified requirements or user expectations [1]. These defects arise from errors in code, logic, or system behavior, leading to unintended outcomes or performance issues [2]. Predicting software failures is a crucial aspect of software engineering, as it enables early identification of defective modules during the software development life cycle. This early identification contributes to increased software reliability and enhanced quality [3]. As software systems grow in complexity, the occurrence of defects becomes more likely, making it essential to establish mechanisms for minimizing and controlling software faults [4].

Ensuring the delivery of high-quality software requires reducing the presence of defects in the final product. Early detection and correction of defects are cost-effective strategies that not only improve software performance but also reduce the need for extensive rework [5]. Effective defect prediction models help in identifying high-risk components during the early stages of development, ensuring optimal use of testing resources and facilitating proactive quality assurance efforts [6]. Without reliable prediction methods, organizations face challenges such as prolonged debugging efforts, increased maintenance costs, and potential system failures [7]. Therefore, the development and refinement of accurate defect prediction models play a vital role in modern software engineering practices [8].

One of the essential tools for building predictive models in software quality assurance is the use of predictive failure metrics. These metrics serve as indicators of potential defects and provide valuable data for constructing statistical or machine learning models that can estimate the defect-proneness of software modules [9]. Predictive metrics are generally classified into two categories: code metrics and process metrics [10]. Code metrics measure various static characteristics of the source code such as complexity, lines of code, cohesion, and coupling. In contrast, process metrics capture information related to software development and evolution, such as change frequency, revision history, and developer activity patterns [11]. Using these predictive metrics, software engineers can develop models that anticipate defect occurrence in specific components, allowing them to implement proactive testing and preventive strategies. This predictive capability is especially valuable in large-scale and safety-critical systems, where even minor defects can have serious implications [12].

Many organizations now rely on such metrics to assess software reliability and drive continuous improvement initiatives in software quality management [13]. The ability to use these metrics effectively determines the performance of defect prediction models and ultimately contributes to reducing the number of post-release defects [14]. The relationship between software metrics and defect occurrence has been widely studied using various analytical and computational approaches. Early efforts primarily utilized traditional statistical techniques, such as logistic regression and discriminant analysis, to identify key defect predictors [15]. These methods, though useful, often struggled with handling non-linear relationships and required extensive preprocessing and manual feature selection.

To address these limitations, machine learning algorithms have been increasingly adopted due to their ability to automatically learn complex patterns and relationships from historical defect data [16]. Machine learning has emerged as a powerful approach for software defect prediction, enabling the creation of data-driven models that can classify software modules as defective or non-defective based on various features. Techniques such as decision trees, support vector machines (SVM), neural networks, and ensemble methods have been applied with promising results in this area [17]. These models outperform traditional methods in many cases by learning from historical defect data and generalizing to new, unseen code modules. For example, decision trees are known for their interpretability and ease of implementation, while support vector machines offer strong performance in high-dimensional spaces [18].

With the increasing availability of large software repositories and defect databases, machine learning models can now be trained on real-world data, improving their robustness and practical applicability. The integration of these models into software development pipelines enables continuous defect prediction and real-time feedback to developers, facilitating a more proactive approach to quality assurance [19]. Furthermore, modern machine learning methods can adapt to changing project dynamics and incorporate new metrics as software evolves, offering flexibility and scalability that traditional approaches lack [20].

Despite the progress made in machine learning-based defect prediction, there are still challenges, especially when it comes to class imbalance. In many real-world software projects, defective modules represent a small fraction of the total codebase, leading to a skewed distribution in the dataset. Traditional classifiers may fail to recognize minority class examples, resulting in high false negatives. Cost-sensitive learning has been proposed as a solution to this issue, where models are trained to assign higher penalties to misclassifying defective modules [2]. This approach improves the sensitivity of models in identifying actual defects, thereby enhancing prediction performance in imbalanced datasets.

Cost-sensitive boosting and ensemble models have shown substantial potential in improving software defect prediction. These models combine multiple base learners to create a stronger overall model that better captures the complexity of the data. For instance, Zheng [2] proposed a cost-sensitive boosting neural network model that demonstrated superior performance in detecting software defects, particularly in imbalanced settings. Similarly, Liu et al. [5] introduced a two-stage cost-sensitive learning framework that improved predictive accuracy while minimizing false positives. Such advancements indicate the growing importance of cost-sensitive strategies in modern defect prediction systems.

In addition to cost-sensitive learning, capture-recapture models have also been utilized for estimating the total number of defects in a software system. Originally used in biological population studies, capture-recapture techniques have been adapted to software engineering to estimate defect content by analyzing overlap between defect detection efforts. Briand et al. [3] and Runeson and Wohlin [6] evaluated the effectiveness of various capture-recapture models and found them useful for early defect estimation, especially in inspection-based quality assurance processes. While these models are not predictive in the machine learning sense, they provide valuable insights into the expected number of undetected defects and can complement predictive models.

As research in this domain progresses, ensemble learning techniques like Random Forests have gained popularity due to their robustness and high accuracy in classification tasks. A Random Forest consists of multiple decision trees that work together to produce a consensus prediction, thus reducing overfitting and improving generalization. Bernard et al. [10] explored how decision tree selection affects the performance of Random Forests and suggested strategies for optimizing ensemble construction. These methods have also been extended to the domain of intrusion detection, demonstrating their versatility and adaptability across different problem areas.

Intrusion detection systems (IDS), although different in application from defect prediction, share similarities in terms of the underlying classification problem. IDS aim to identify unauthorized or malicious activity within a network or system, often using the same machine learning algorithms applied in defect prediction. Various researchers have investigated the use of ensemble learning, feature selection, and dimensionality reduction techniques to improve IDS performance. For instance, Tesfahun and Bhaskari [11] utilized Random Forests with Synthetic Minority Oversampling Technique (SMOTE) and feature reduction to enhance intrusion detection accuracy. Likewise, Le et al. [12] demonstrated that Principal Component Analysis (PCA) could enhance the performance of deep learning models like Gated Recurrent Units (GRU) in detecting anomalies in network traffic

**Support Vector Machines (SVM)**: SVM is particularly effective for binary classification problems and can create a robust decision boundary between defect-prone and non-defect-prone classes. However, it can be computationally expensive for large datasets and may not perform well with non-linearly separable data without kernel transformations.

**Naïve Bayes**: This is a probabilistic model based on Bayes' theorem, which is relatively simple to implement and effective with high-dimensional datasets. It assumes independence among features, which might not hold true in all software metrics datasets, sometimes limiting accuracy.

**Decision Trees**: Decision Trees are popular due to their interpretability and ability to handle both categorical and continuous variables. They work by recursively splitting the data based on feature values, which makes them simple to understand but prone to overfitting, especially with small datasets.

**Random Forest**: An ensemble of Decision Trees, Random Forest is known for improved accuracy and generalization by averaging multiple trees. This approach reduces the risk of overfitting while providing reliable predictions in complex datasets.

**Random Forest and Gradient Boosting:** Both are ensemble techniques that combine multiple decision trees to increase prediction reliability and accuracy. Random Forest reduces overfitting by creating many trees using random subsets of the data, while Gradient Boosting refines predictions by iteratively correcting errors from previous trees, resulting in high accuracy.

**AdaBoost and XGBoost:** These methods are variations of boosting techniques that adjust weights for misclassified instances, improving focus on difficult-to-predict defects. XGBoost, in particular, is known for its speed and performance, making it popular in defect prediction studies.

**Neural Networks:** These models can capture complex, non-linear relationships in data, making them suitable for defect prediction when ample data is available. Neural networks learn feature interactions automatically, which helps uncover hidden patterns but requires significant computational power.

**Recurrent Neural Networks (RNNs):** RNNs and their variations (e.g., LSTM) are used to analyze time- sequential data, like code change history, to detect evolving patterns in software defects.

**Class Imbalance:** In many datasets, the majority of code modules are defect-free, resulting in imbalanced data. This imbalance can lead models to favor the majority class, which decreases the effectiveness of defect prediction. Techniques like oversampling, undersampling, and Synthetic Minority Over-sampling Technique (SMOTE) help address this issue.

**Data Quality:** Defect prediction models require large amounts of high-quality data. Noisy data or mislabeled defects reduce model accuracy, making data preprocessing crucial.

**Model Interpretability:** Advanced machine learning models, especially neural networks, often function as “black boxes” where the internal reasoning is difficult to interpret. For practical adoption, software engineers need interpretable predictions that provide actionable insights.

**Generalization:** Different software projects may have unique characteristics, and models trained on one project dataset might not generalize well to another. Cross-project defect prediction remains a difficult problem, and transfer learning approaches are being explored to address this.

**Overfitting:** Overfitting occurs when models capture noise rather than meaningful patterns. Regularization techniques, cross-validation, and ensemble methods help prevent overfitting, especially in complex models.

**Future Research Directions**

To further advance defect prediction, current research is focusing on:

**Transfer Learning**: This approach seeks to leverage knowledge from one project’s defect data to improve predictions in another, enhancing generalization across diverse software projects.

**Explainable AI (XAI):** Developing interpretable models can help practitioners understand predictions and trust model outcomes, fostering real-world adoption.

**Incremental Learning:** For projects with evolving datasets, incremental learning methods allow models to adapt as new data becomes available, maintaining predictive accuracy over time.

**Automated Machine Learning (AutoML):** AutoML aims to automate the model selection and tuning process, making defect prediction more accessible for practitioners who may not be machine learning experts.

**Practical Implications and Benefits**

Implementing defect prediction models provides tangible benefits for software projects:

**Cost Reduction:** By focusing testing efforts on defect-prone areas, organizations reduce the time andresources required for comprehensive testing.

**Improved Software Quality:** Early defect detection leads to higher code quality, ultimately resulting in a better product for end-users.

**Risk Management:** By predicting potential defects, managers can make informed decisions about release timelines, resource allocation, and mitigation strategies.

**Enhanced Team Efficiency:** Developers can focus their efforts on critical areas, leading to faster bug resolution and fewer disruptions in development.

In summary, software defect prediction represents a powerful application of machine learning with far-reaching benefits for software development. As research continues to advance, integrating sophisticated

ML techniques into real-world software development processes promises to further.

Software defect prediction remains a key strategy for improving software quality and reliability. By combining statistical and machine learning models with metrics-based analysis, organizations can predict and address potential issues proactively, reduce rework, and deliver more reliable software. With the advancement of new techniques like deep learning and transfer learning, defect prediction is becoming more accurate and adaptable, making it a powerful tool in the software engineering process.

Individuals and society increasingly rely on advanced software systems. Because software is intertwined with all aspects of our lives, it is essential to produce reliable and trustworthy systems economically and quickly. In order to ensure the desired software quality at a lower cost, much effort has been invested on software reliability and software quality assurance (SQA) . With limited resources, however, this is increasingly being challenged by the rapid growth in size and complexity of today’s software. Defective software modules increase the development and maintenance costs and cause customer dissatisfaction .

Software defect prediction is one of the SQA activities that aims to automatically predict fault-p;./prone software modules using historical software information from an earlier deployment or identical objects, for example source code edit logs and bug reports , before the actual testing process begins. Effective defect prediction could help test managers locate bugs and facilitate the allocation of limited SQA resources optimally and economically; thus, it has become an extremely important research topic. Commonly, a prediction model is used to predict the defective software modules in one of the three categories binary class classification of defects, number of defects/defect density prediction , and severity of defect prediction .

Among them, the binary class classification is the most frequently used types of prediction scheme, where software modules having one or more defects are marked as defected and modules having zero defects are marked as non-defected. In this type of defect prediction schema, researchers have explored the use of various classification techniques, including statistical techniques such as Naïve Bayes (NB) and Logistic Regression ; supervised techniques such as Decision Tree (DT) , Support Vector Machine (SVM) , ensemble methods , and Case Based Reasoning; semi supervised techniques such as Expectation Maximization (EM) ; and unsupervised techniques such as K-means clustering and Fuzzy clustering . Most of the studies in the literature have used statistical and supervised learning techniques .

Although a large number of studies have been conducted to build and evaluate defect prediction models using different classification techniques in the context of binary class classification, still the prediction accuracy of defect prediction techniques is found to be considerably low, with a high misclassification rate . Looking at these results, one questions the dependability of these techniques for software defect prediction . Therefore, it will be important to design more advanced techniques to improve the performance of defect prediction models .

In this work, a hybrid heterogeneous ensemble approach is proposed for improving the accuracy of software defect prediction. The core argument for this approach is to develop expert and robust classification models of different natures based on groups of similar points. In other words, the classification models are of different machine learning types like lazy classifiers, decision trees, naïve bayes, and ensembles.

While on the other hand, similar points refer to a group of points that are as close as possible according to a similarity measure like the euclidean measure. These groups of data are generated using a clustering stage. Unlike most of the previous works that generate general models for all data, this work aims to develop several expert models based on the characteristics of the data. Two versions of the proposed approach are developed and experimented. The first is based on simple classifiers (i.e., k-Nearest Neighbour (*k*-NN), NB, and DT), and the second is based on ensemble ones (i.e., Bagging, Adaptive Boosting (AdaBoost), Random Forest (RF), and XGBoost (XGB)). Extensive experiments based on 21 well-known benchmark datasets are conducted to evaluate the proposed approach.

The remainder of this article is organized as follows: The next section presents related work on defect prediction. The preliminaries of the algorithms utilized in the proposed approach are given in presents the proposed hybrid heterogeneous ensemble approach for software defect prediction. discusses the model evaluation metrics, and presents the benchmark datasets specifications. is devoted to the benchmarking experiments and discusses their respective results. Finally, draws conclusions and describes promising directions for future work.

There are a great variety of studies which have developed and applied statistical and machine learning based models for defect prediction in software systems. Basili et al. (1996) [1]have used logistic regression in order to examine what the effect of the suite of object-oriented design metrics is on the prediction of fault- prone classes. Khosh go ftaaret al. (1997)[7] have used the neural network in order to classify the mod-ules of large telecommunication systems as fault-prone or not and compared it with a non-parametric discriminant model.The results of their study have shown that compared to the non-parametric discriminant model, the predictive accuracyof the neural network model had a better result. Then in2002 [6], they made a case study by using regression trees toclassify fault-prone modules of enormous telecommunication systems. Fenton et al. (2002) [4] have used Bayesian BeliefNetwork in order to identify software defects. However, thismachine learning algorithm has lots of limitations whichhave been recognized by Weaver(2003) [14] and Ma et al.(2007) [9]. Guo et al. (2004) [5] have applied RandomForest algorithm on software defect dataset introduced byNASA to predict fault-prone modules of software systemsand compared their model with some statistical and machinelearning models. The result of this comparison has shownthat compared to other methods, the random forest algorithmhas given better predictive accuracy. Ceylan et al. (2006) [2]have proposed a model which uses three machine learningalgorithms that are Decision Tree, Multilayer Perceptron andRadial Basis Functions in order to identify the impact ofthis model to predict defects on different software metricdatasets obtained from the real\*life projects of three big-sizesoftware companies in Turkey

Software defect prediction plays an increasingly vital role in the realm of software development, as it significantly contributes to enhancing software quality while simultaneously reducing the time and costs associated with software testing processes, which are often critical in ensuring that software products meet the necessary standards and requirements before they are released to the market. The advent of machine learning has revolutionized this field, as it focuses on the development of sophisticated computer programs that possess the remarkable ability to teach themselves, enabling them to grow and adapt when exposed to new data. This self-improving capability of machines allows them to enhance their performance based on previous outcomes, thereby making them invaluable tools in the software defect prediction landscape.

To achieve effective software defect prediction, various machine learning techniques are employed to preprocess and refine the data, specifically aimed at removing unnecessary and erroneous information from the dataset, which can otherwise skew results and lead to inaccurate predictions. The significance of software defect prediction becomes even more pronounced when planning a software project, as it allows project managers and developers to allocate resources more effectively and focus their efforts on areas of the software that are more prone to defects, thus enhancing overall project efficiency. The complexity of this problem necessitates a greater effort in leveraging software metrics and defect datasets, which serve as essential tools in the prediction process. Metrics, in this context, refer to the quantitative measures that establish a relationship between numerical values and their application within the software, making them instrumental in predicting potential defects. The primary goal of this survey paper is to provide a comprehensive understanding of the existing techniques employed for predicting software defects, highlighting the various methodologies and approaches that have been developed and their effectiveness in real-world applications. The results obtained from the proposed approaches indicate a marked improvement in efficiency and accuracy when compared to traditional techniques such as Support Vector Machines (SVM), Naïve Bayes, and Decision Trees, which have been widely used in the past for defect prediction tasks. Specifically, the results derived from the proposed method reveal impressive performance metrics, with a performance time of just 3.24 minutes, an accuracy rate of 96.78%, and a remarkably low error rate of 0.21%, underscoring the efficacy of the machine learning techniques employed in this study. These findings not only demonstrate the potential of advanced machine learning algorithms in the domain of software defect prediction but also emphasize the need for ongoing research and development in this area to further refine and enhance predictive capabilities. As software systems continue to grow in complexity and scale, the demand for reliable and efficient defect prediction methods will only increase, making it imperative for developers and researchers to explore innovative solutions that leverage the latest advancements in machine learning and data analysis. By continuing to improve the accuracy and efficiency of software defect prediction, the software development industry can expect to see significant reductions in the time and resources spent on testing and debugging, ultimately leading to higher quality software products that meet the expectations of users and stakeholders alike. In conclusion, the integration of machine learning into software defect prediction represents a significant advancement in the field, providing developers with powerful tools to identify and address potential issues before they escalate, thereby ensuring that software products are not only functional but also reliable and robust in meeting the needs of their intended users.

Developing a software system is an arduous process which contains planning, analysis, design, implementation, testing, integration and maintenance. A software engineer is expected to develop a software system on time and within limited the budget which are determined during the planning phase. During the development process, there can be some defects such as improper design, poor functional logic, improper data handling, wrong coding, etc. and these defects may cause errors which lead to rework, increases in development and maintenance costs decrease in customer satisfaction. A defect management approach should be applied in order to improve software quality by tracking of these defects. In this approach, defects are categorized depending on the severity and corrective and preventive actions are taken as per the severity defined. Studies have shown that ’defect prevention’ strategies on behalf of ’defect detection’ strategies are used in current methods [10]. Using defect prevention strategies to reduce defects generating during the software development the process is a costly job. It requires more effort and leads to increases in project costs. Accordingly, detecting defects in the software on the front line of the project life cycle is crucial. The implementation of machine learning algorithms which is the binary prediction model enables identify defect- prone modules in the software system before a failure occurs during development process. In this research, our aim is to evaluate the software defect prediction performance of seven machine learning algorithms by utilizing quality metrics; accuracy, precision, recall, F-measure associated with defects as an independent variable and find the best category while comparing software defect prediction performance of these machine learning algorithms within the context of four NASA datasets obtained from public PROMISE repository [12]. The selected machine learning algorithms for comparison are used for supervised learning to solve classification problems. They are two tree- structured classifier techniques: (i) Bagging and Random Forests (RF); two neural networks techniques: (i) Multilayer Perceptron (MLP) and (ii) Radial Basis Function (RBF); two Bayesian classifier techniques: (i) Naive Bayes and (ii) Multinomial Naive Bayes; and one discriminative classifier Support Vector Machine (SVM). The remainder of the paper is organized as follows: Section 2 briefly describes the related work, while Section 3 describes the experimental methodology in detail. Section 4 contains the conclusion of the experimental study and underlined some possible future research directions.

There are a great variety of studies which have developedand applied statistical and machine learning based models fordefect prediction in software systems. Basili et al. (1996) [1]have used logistic regression in order to examine what theeffect of the suite of object-oriented design metrics is on theprediction of fault-prone classes. Khoshgoftaar et al. (1997)[7] have used the neural network in order to classify the mod-ules of large telecommunication systems as fault-prone or notand compared it with a non-parametric discriminant model.The results of their study have shown that compared to thenon-parametric discriminant model, the predictive accuracy of the neural network model had a better result. Then in2002 [6], they made a case study by using regression trees to classify fault-prone modules of enormous telecommunication systems. Fenton et al. (2002) [4] have used Bayesian Belief Network in order to identify software defects. However, this machine learning algorithm has lots of limitations which have been recognized by Weaver(2003) [14] and Ma et al.(2007) [9]. Guo et al. (2004) [5] have applied Random Forest algorithm on software defect dataset introduced by NASA to predict fault-prone modules of software systems and compared their model with some statistical and machine learning models. The result of this comparison has shown that compared to other methods, the random forest algorithm has given better predictive accuracy. Ceylan et al. (2006) [2]have proposed a model which uses three machine learning algorithms that are Decision Tree, Multilayer Perceptron and Radial Basis Functions in order to identify the impact of this model to predict defects on different software metric datasets obtained from the real\*life projects of three big-size software companies in Turkey. all of the machine learning algorithms had similar results which have enabled to predict potentially defective software and take actions to correct them. Elish et al. (2008) [3]have investigated the impact of Support Vector Machines on four NASA datasets to predict defect-proneness of software systems and compared the prediction performance of SVM against eight statistical and machine learning models. The results have indicated that the prediction performance of SVM has been much better than others. Kim et al. (2011) [8]have investigated the impact of the noise on defect predictionto cope with the noise in defect data by using a noisedetection and elimination algorithm. The results of the studyhave presented that noisy instances could be predicted withreasonable accuracy and applying elimination has improvedthe defect prediction accuracy. Wang at all. (2013) [13] haveinvestigated re-sampling techniques, ensemble algorithms andthreshold moving as class imbalance learning methods forsoftware defect prediction. They have used different methodsand among them, AdaBoost.NC had better defect predictionperformance. They have also improved the effectiveness andefﬁciency of AdaBoost.NC by using a dynamic version ofit. Ren at al. (2014) [11] have proposed a model to solvethe class imbalance problem which causes a reduction in theperformance of defect prediction. The Gaussian function hasbeen used as kernel function for both the Asymmetric KernelPartial Least Squares Classiﬁer (AKPLSC) and AsymmetricKernel Principal Component Analysis Classiﬁer (AKPCAC)and NASA and SOFTLAB datasets have been used forexperiments. The results have shown that the AKPLSC hadbetter impact on retrieving the loss caused by class imbalanceand the AKPCAC had better performance to predict defectson imbalanced datasets. There is also a systematic reviewstudy conducted by Malhotra to review the machine learningalgorithms for software fault prediction.

As software development continues to evolve, the need for effective defect prediction will only increase. The growing complexity of software systems, coupled with the demand for rapid delivery, underscores the importance of identifying and addressing defects early in the development process. By embracing machine learning and continuously refining their predictive models, organizations can enhance their ability to deliver high-quality software products that meet user expectations and withstand the pressures of an ever-changing technological landscape.

The integration of machine learning into software defect prediction presents both significant opportunities and challenges. Organizations that successfully harness this technology will not only improve their software quality but also position themselves as leaders in the competitive software development landscape. By investing in research, collaboration, and ethical practices, the industry can pave the way for more effective and responsible use of machine learning in defect prediction, ultimately leading to better software outcomes for all stakeholders involved. The journey toward achieving excellence in software quality through predictive modeling is ongoing, and the potential for innovation in this field remains vas the integration of machine learning into software defect prediction represents a transformative advancement in the field of software engineering. By leveraging data-driven approaches, organizations can proactively identify and address potential defects, significantly enhancing software quality and reliability. As the complexity of software systems continues to grow and the demand for rapid delivery intensifies, effective defect prediction becomes increasingly essential.

However, the successful implementation of machine learning in defect prediction is not without its challenges. Issues such as data quality, model interpretability, and ethical considerations must be carefully navigated to ensure that predictive models are both accurate and responsible. Organizations must foster a culture of collaboration and continuous improvement, encouraging cross-functional teams to work together in refining these models and integrating them into existing workflows.

Future research and development in this area hold great promise, with opportunities to explore hybrid

models, natural language processing, transfer learning, and explainable AI. By addressing these areas, the software engineering community can enhance the robustness and applicability of defect prediction models across diverse contexts and programming environments.

Ultimately, organizations that embrace machine learning for defect prediction will not only improve their software quality but also gain a competitive edge in the rapidly evolving technology landscape. By prioritizing innovation, ethical practices, and a commitment to continuous learning, the industry can ensure that the benefits of predictive modeling are realized, leading to better software outcomes and enhanced user satisfaction. The journey toward excellence in software quality through machine learning is ongoing, and its potential to reshape the future of software development is both exciting and significant.

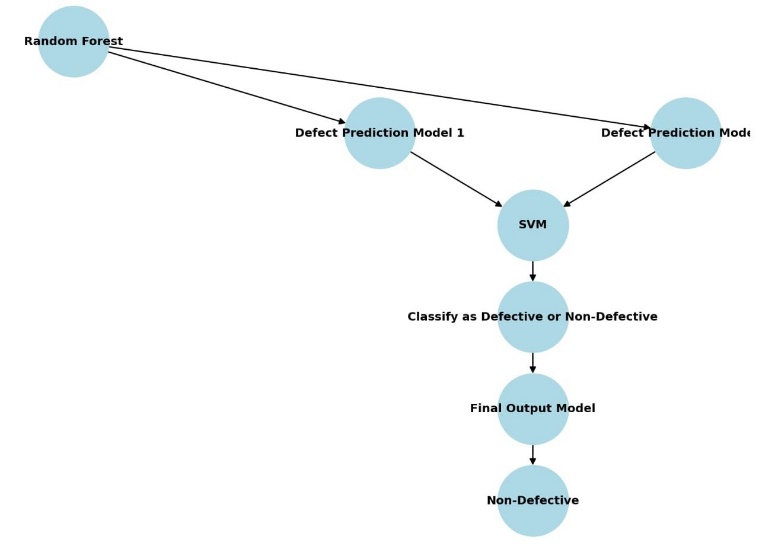
As we look to the future, the evolution of software defect prediction through machine learning will likely be characterized by several key trends and developments. First, the increasing availability of large datasets will provide a wealth of information for training more sophisticated models. As organizations collect more data from various stages of the software development lifecycle, including code repositories, issue tracking systems, and user feedback, the potential for creating highly accurate and context-aware defect prediction models will expand. This abundance of data will enable the development of models that not only predict defects more accurately but also provide insights into the underlying causes, allowing teams to implement preventative measures.

Moreover, the adoption of cloud-based tools and platforms for software development will facilitate easier access to advanced machine learning capabilities. Moreover, the increasing emphasis on user experience in software development highlights the need for defect prediction models that consider not just technical metrics but also user behavior and feedback. Understanding how users interact with software can provide valuable insights into potential areas of failure. By analyzing user engagement data, organizations can develop models that predict defects based on real-world usage patterns, ultimately leading to more reliable and user-friendly software. This user-centric approach fosters a closer alignment between development teams and end-users, ensuring that the software meets the needs and expectations of those who rely on it.

The ethical implications of machine learning in defect prediction are also becoming more pronounced. As organizations leverage data to inform their predictive models, they must ensure that they are doing so responsibly and transparently. Establishing ethical guidelines for data usage, model development, and decision-making processes will be essential in building trust among stakeholders. This commitment to ethical practices will not only enhance the credibility of predictive models but also promote a culture of accountability within organizations.

In addition to ethical considerations, the importance of collaboration across disciplines cannot be overstated. The development of effective defect prediction models requires input from various stakeholders, including software engineers, data scientists, quality assurance professionals, and business leaders. By fostering an environment of collaboration, organizations can ensure that their predictive models are well-rounded, taking into account diverse perspectives and expertise. This interdisciplinary approach will lead to more robust models that are better equipped to address the complexities of software development.

**General architecture :**



**Figure 1****: General architecture**

In Figure 1flowchart illustrates the process of software defect prediction using machine learning techniques. The approach begins with the Random Forest model, which is utilized for feature selection and preliminary defect classification. The output from the Random Forest model is then passed into two different Defect Prediction Models, each employing distinct methodologies to enhance predictive accuracy.

The results from these models are subsequently processed by a Support Vector Machine (SVM), a supervised learning algorithm that classifies software modules based on defect likelihood. The SVM model determines whether a given software component should be classified as defective or non-defective, based on extracted features and learning patterns.

Following this classification, the prediction undergoes a final evaluation in the Final Output Model, ensuring refined accuracy in defect classification. If the software component is classified as non-defective, it is deemed suitable for deployment. Conversely, defective modules require further analysis and corrective actions before release.

This flowchart highlights a machine learning-based framework that enhances software defect prediction by integrating Random Forest for feature extraction and SVM for classification. The approach improves software quality assurance, minimizes failure risks, and optimizes defect management in the software development lifecycle.

# CHAPTER 2

**REQUIREMENT ANALYSIS**

**2.1 Literature Survey**

A literature survey on software defect estimation explores various approaches, methods, and models used to predict or estimate the number of defects that may exist or appear in software systems. Software defect estimation is a critical aspect of software quality assurance and helps organizations plan resources, reduce costs, and improve product reliability. Accurate defect estimation enables project managers to anticipate potential issues, allocate resources effectively, and optimize testing and debugging efforts.

In recent years, researchers have developed various techniques for defect estimation, leveraging statistical, machine learning, and artificial intelligence approaches. Traditional methods often rely on historical data and statistical models, while newer techniques incorporate machine learning algorithms that can adapt to specific project characteristics and learn from vast datasets. Key areas of focus in the literature include defect prediction models, defect density analysis, code quality metrics, and the impact of different development methodologies on defect rates.

This literature survey provides a comprehensive overview of these techniques, examining studies that address key factors influencing defect prediction accuracy, such as data quality, model selection, and evaluation metrics. It also highlights the limitations of existing methods and the potential for future research to develop more robust, scalable, and accurate defect estimation solutions.

Table 1. Literature Survey on A NOVEL MACHINE LEARNING METHOD FOR SOFTWARE DEFECT ESTIMATION

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| S.No | Authors | Title name | Description | Year |
| 1 | Diana-Lucia Miholca, Vlad- Ioan Tomescu, Gabriela Czibula | HEDF: A  Method for EarlyForecastin g Software Defects Based on Human Error Mechanisms | To analyze the impact of software features on the performance of deep learning-based software defect prediction models. | 2022 |
| 2 | Diana-Lucia Miholca, Vlad-Ioan Tomescu, Gabriela Czibula | An in-Depth Analysis of the Software Features’ Impact on the Performance of Deep Learning- Based Software Defect Predictors | To analyze the impact of software features on the performance of deep learning-based software defect prediction models. | 2022 |
| 3 | Yahaya Zakariyau Bala, Pathiah | Improving Cross- Project software | To improve cross-project software defect prediction (CPSD) performance by addressing distribution differences and high- | 2022 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Abdul Samat, Khaironi Yatim Sharif, Noridayu Manshor | Defect Prediction Method Through Transformati on and Feature Selection  Approach | dimensional features. |  |
| 4 | Maohua Gan, Zeynep Yücel, Akito Monden | HOPE:  Software Defect Prediction Model Constructio n Method via Homomorp hic Encryption | To address class imbalance issues in evaluating software defect prediction models by proposing neg/pos-normalized accuracy measures.. | 2023 |
| 5 | Ahmad Muhaim in Ismail, Siti Hafizah Ab Hamid, Asmiza Abdul Sani, Nur Nasuha Mohd  Daud | Toward Reduction in False Positives Just-In- Time Software Defect Prediction Using Deep Reinforcement Learning | To reduce false positives in just- in-time (JIT) software defect prediction, enhancing prediction reliability using Deep Q-Network (DQN) based reinforcement learning models. | 2024 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | Iqra Mehmood, Sidra Shahid, Hameed Hussain, Inayat Khan, Shafiq Ahmad, Shahid Rahman, Najeeb  Ullah, | A Novel Approach to Improve Software Defect Prediction Accuracy Using Machine Learning | improve software defect prediction accuracy using machine learning techniques by integrating feature selection for better classifier | 2023 |
| 7 | Jie Zhang, Jiajing Wu, Chuan Chen, Zibin Zheng, Michael R. Lyu | CDS: A  Cross- Version Software Defect Prediction Model With Data Selection and class  overlapping | A address the challenges of Cross- Version Defect Prediction (CVDP) by solving data distribution differences | 2020 |
| 8 | Li Jiang, Shujuan Jiang, Lina Gong, Yue Dong, Qiao Yu | Which Process Metrics Are Significantly Important to Change of Defects in Evolving Projects: An Empirical  Study | To explore which process metrics are significantly important to defect state changes in evolving software projects | 2020 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 10 | Jiwon Choi, Taeyoung Kim, Duksan Ryu, Jongmoon Baik, Suntae Kim | Just-in-Time Defect Prediction for Self- driving Software via a Deep Learning Model | To develop an effective Just-in- Time (JIT) defect prediction model for edge computing applications, particularly for self-driving software. | 2023 |
| 11 | Wanzhi Wen, Chenqiang Shen, Xiaohong Lu, Zhixian Li, Haoren Wang, Ruinian Zhang, Ningbo  Zhu | Cross- Project Software Defect Prediction Based on Class Code Similarity | To improve software defect prediction accuracy across different projects by addressing data distribution differences. | 2022 |
| 12 | Misbah Ali, Tehseen Mazhar, Yasir Arif, Shah A. Al-Otaibi, Yazeed Yasin Ghadi, Tariq Shahzad, Muhamma d Amir Khan, Habib  Hamam | Software Defect Prediction Using an Intelligent Ensemble- Based Model | Improve software quality and reduce testing costs by predicting defects | 2024 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 13 | Jaewook Lee, Jiwon Choi, Duksan Ryu, Suntae Kim | Holistic Parameter Optimization for Software Defect Prediction | Improve the accuracy of software defect prediction by optimizing all parameters throughout the process. | 2022 |
| 14 | Petar Afric, Davor Vukadin, Marin Silic, Goran Delac | Empirical Study: How Issue Classificatio n Influences Software Defect Prediction | Examine how different issue classification methods impact the quality of software defect prediction datasets and model performance. | 2023 |
| 15 | Examine how different issue classificati on methods impact the quality of software defect prediction datasets and model performanc  e. | Systematic Mapping: Artificial Intelligence Techniques in Software Engineering | Map and review the use of AI techniques across all phases of software engineering. | 2022 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 16 | Lucija Šikić, Petar Afrić, Adrian Satja Kurdiija, Marin Šilić | Improving Software Defect Prediction by Aggregated Change Metrics | Enhance defect prediction by incorporating aggregated change metrics into the model | 2021 |
| 17 | Dimah Al- Fraihat, Yousef Sharrab, Abdel- Rahman Al- Ghuwairi, Hamzeh Alshishani,  Abdulmohs  en Algarn | Hyperparam eter Optimization for Software Bug Prediction Using Ensemble Learning | To enhance software bug prediction accuracy and reduce costs using ensemble learning and hyperparameter optimization. | 2024 |
| 18 | Jianming Zheng, Xingqi Wang, Dan Wei, Bin Chen, Yanli Shao | A Novel Imbalanced Ensemble Learning in Software Defect Prediction | To improve predictive accuracy for minority classes in imbalanced software defect datasets. | 2021 |
| 19 | Mansi Gupta, Kumar Rajnish, Vandana Bhattacharj ee | Cognitive Complexity and Graph Convolution al Approach Over Control Flow Graph for Software Defect Prediction | To improve software defect prediction using cognitive complexity measures and graph convolutional networks (GCNs). | 2022 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 20 | Qusay Idrees Sarhan, Árpád Beszédes | A Survey of Challenges in Spectrum- Based Software Fault Localization | To identify, categorize, and discuss the challenges and issues associated with Spectrum-Based Fault Localization (SBFL). | 2022 |
| 21 | Maohua Gan, Zeynep Yücel, Akito Monden | Neg/pos- Normalized Accuracy Measures for Software Defect Prediction | To address the class imbalance issue in assessing the performance of software defect prediction models. | 2022 |
| 22 | Lei Qiao a,  Xuesong Li a,  Qasim Umer a,  Ping Guo b | Deep learning based software defect prediction | The aim of this study is to develop a deep learning-based approach for software defect prediction to improve the accuracy and efficiency of defect identification in software development. | 2022 |
| 23 | Vishal Giraddi\*, Shantala Giraddi, Narayan D G, Anupama Bidaragaddi, Suvarna G  Kanakared d | Machine Learning Approach to Intrusion Detection: Performance  Evaluation | Evaluate the performance of machine learning approaches for intrusion detection in computer networks | 2020 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 24 | Tarunim Sharmaa , Aman Jatainb , Shalini Bhaskarc and Kavita Pabrejad | Ensemble Machine Learning Paradigms in Software Defect Prediction | Develop an ensemble machine learning approach for software defect prediction to improve accuracy. | 2021 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | | | | |
|  | 25 | Zhenyang Sun , Gangyi An , Yixuan Yang , Yasong Liu \* | Optimized machine learning enabled intrusion detection 2  system for Internetof medical things | Create a smart system to detect hacking in medical devices connected to the internet. | 2023 |
| 26 | Vishal Giraddi\*, Shantala Giraddi, Narayan D G, Anupama Bidaragaddi, Suvarna G Kanakareddi | Machine Learning Approach to Intrusion Detection: Performance  Evaluation | Evaluate the performance of machine learning approaches for intrusion detection in computer networks | 2020 |
| 27 | Tarunim Sharmaa , Aman Jatainb  , | Ensemble Machine Learning Paradigms in | Develop an ensemble machine learning approach for software defect prediction to improve accuracy. | 2021 |
| 28 | Ali Waqas a,\*  , Mohamad T. Araji b | Machine learning-aided thermography for autonomous heat loss detection  in buildings | Develop a machine learning-aided thermography system for autonomous heat loss detection in buildings | 2020 |
| 29 | Imane Bayane \*  , John Leander, Raid Karoumi | An unsupervised machine learning approach for real-time damage  detection in bridges | Develop an unsupervised machine learning approach for real-time damage detection in bridges using vibration data | 2023 |
| 28 | | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | | | | |
|  | 30 | Vasilis Belis, Patrick Odagiu, | Machine learning for anomaly detection in particle physics | Develop machine learning-based anomaly detection methods for identifying new physics beyond the Standard Model in particle physics. | 2022 |
| 31 | Rajesh Natarajan a, Santosh Reddy P b, SubashChan | Fault detection and state estimation in robotic automatic | Develop a machine learning-based approach for fault detection and state estimation in robotic automatic control | 2020 |
| 32 | amad Alsawalqah, Neveen Hijazi  Mohammed Eshtay Hossam Faris Ahmed Al Radaideh Ibrahim Aljarah Yazan Alshamaileh | Software Defect Prediction Using Heterogeneous Ensemble Classification Based on Segmented Patterns | Heterogeneous Ensemble Classification Based on Segmented Patterns | 2020 |
| 33 | Revoori Veeharika Reddy, Nagella Kedharnath, Mandi Akif Hussain, S.  Vidya | Software Defect Estimation using Machine Learning Algorithms | Defecting the software of the machine | 2021 |
| 34 | Abdullah Alsaeedi, Mohammad Zubair Khan | Software Defect Prediction Using Supervised Machine Learning and Ensemble Techniques: A  Comparative Study | Prediction of supervised of software defect | 2019 |
| 29 | | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | | | | |
|  | 35 | B. Vahabzade str., 9A, AZ1141 Baku,  Azerbaijan | Reliability of software systems | Reliability of software systems is one of the main indicators of quality. | 2023 |
| 36 | Hamad Alsawalqah  , Neveen Hijazi  , Mohammed Eshtay  , Hossam Faris  \*,  Ahmed Al Radaideh  ,  , Ibrahim Aljarah and Yazan Alshamaileh | Heterogeneous Ensemble Classiﬁcation Based on Segmented Patterns | Software Defect Prediction Using Heterogeneous Ensemble Classiﬁcation Based on Segmented Patterns | 2020 |
| 37 | Ayse Tosun, Burak Turhan, Ayse Bener | Ensemble of Software Defect Predictors: A Case Study | a defect prediction model based on ensemble of classifiers | 2021 |
| 38 | Ashraf Sayed Abdou, Nagy Ramadan Darwish | Early Prediction of Software Defect using Ensemble Learning: A Comparative Study | Prediction of Software Defect using Ensemble Learning: A Comparative Study | 2020 |
| 39 | Vasilis Belis, Patrick Odagiu, | Machine learning for anomaly detection in particle physics | Develop machine learning-based anomaly detection methods for identifying new physics beyond the Standard Model in particle physics. | 2019 |
| 40 | Tarunim Sharmaa , Aman Jatainb | Ensemble Machine Learning Paradigms in | Develop an ensemble machine learning approach for software defect prediction to improve accuracy | 2021 |

### 2.2 Research Methodology:

### This section outlines the research methodology employed in this study to develop a software defect prediction model using machine learning techniques. The methodology consists of several key steps: data collection, data preprocessing, feature selection, model selection, training and testing, and evaluation.

### Data Collection

### The first step in the research methodology involves collecting a suitable dataset for software defect prediction. This dataset should include historical data on software projects, containing both code metrics and defect information. Commonly used datasets in the field include:

### NASA Metrics Data Program: A well-known dataset containing various software metrics and defect data.

### PROMISE Repository: A collection of datasets for software engineering research, including defect prediction datasets.

### Open Source Projects: Publicly available repositories (e.g., GitHub) can also be mined for defect data and corresponding metrics.

### 2. Data Preprocessing:

### Once the dataset is collected, it undergoes preprocessing to ensure its quality and suitability for analysis. The preprocessing steps may include:

### Model Selection

### In this study, various machine learning algorithms will be employed to predict software defects. The selected models may include:

### Support Vector Machines (SVM): A powerful classification technique that works well for high-dimensional data.

### Naïve Bayes: A probabilistic classifier based on Bayes' theorem, suitable for large datasets.

### Decision Trees: A simple yet effective model that provides interpretable results.

### Artificial Neural Networks (ANN): A more complex model that can capture non-linear relationships in the data.

### Cross-Validation: K-fold cross-validation will be employed to ensure that the models are robust and to mitigate overfitting. This involves dividing the training data into K subsets and training the model K times, each time using a different subset for testing.

### Evaluation Metrics:To evaluate the performance of the models, several metrics will be utilized:

### Accuracy: The proportion of correctly predicted instances among the total instances

### Precision: The ratio of true positive predictions to the total predicted positives.

### Recall (Sensitivity): The ratio of true positive predictions to the total actual positives

### F1 Score: The harmonic mean of precision and recall, providing a balance between the two.

### Error Rate: The proportion of incorrect predictions among the total instances.

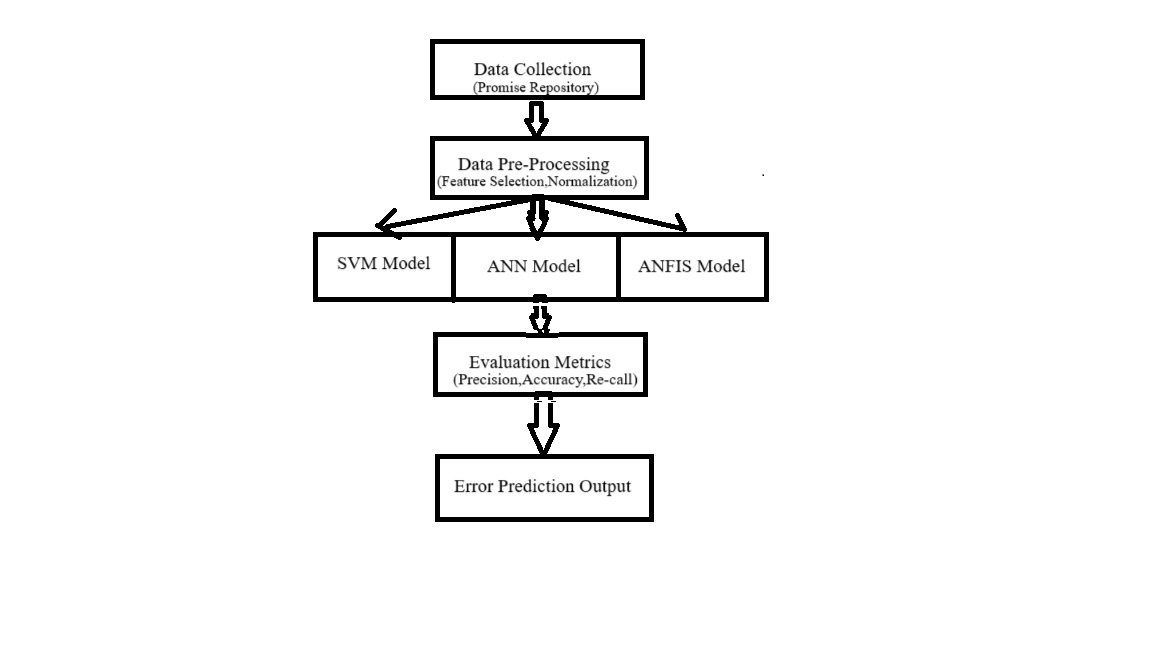
### The proposed method will be compared against traditional techniques (SVM, Naïve Bayes, Decision Trees) to assess its effectiveness. The results will be analyzed to determine which model provides the best accuracy and performance time, as well as the lowest error rate.

# 

**2.3.** **EXISTING SYSTEM:**

The existing system described, you would first need to establish the infrastructure for using the Adaptive Neuro-Fuzzy Inference System (ANFIS) for software fault prediction. The steps below outline a potential approach for implementation based on the provided description.

* The systems which work over the internet suffer from various malicious activities.
* The major problem seen in this field is the intrusion in the system for violating the information.
* Existing results state that there may be some improvements to be done on terms of accuracy and the detection rates and the false alarm rate.
* Some other techniques can replace previously applied techniques such as SVM and Naïve Bayes.
* Also, the study states that the dataset can be improved by using some methods over it.



**Figure 2.3.: Existing system architecture**

In Figure 2**.3** This architecture integrates multiple machine learning models to improve defect estimation accuracy through structured stages:

1. **Data Collection (PROMISE Repository)**
   * Uses the PROMISE Repository, which contains historical defect-related metrics like complexity, coupling, cohesion, and code churn.
2. **Data Preprocessing**
   * Feature selection identifies key attributes for defect prediction.
   * Normalization scales numerical features to improve model performance.
3. **Machine Learning Models**
   * **SVM:** Classifies components as defect-prone or non-defect-prone.
   * **ANN:** Captures complex relationships for better accuracy.
   * **ANFIS:** Combines neural networks and fuzzy logic for adaptability.
4. **Evaluation Metrics**
   * **Accuracy:** Measures correct predictions.
   * **Precision:** Assesses the reliability of defect predictions.
   * **Recall:** Evaluates defect detection capability.
5. **Error Prediction Output**
   * Provides insights to address software defects proactively.
   * Helps reduce maintenance costs and improve reliability.

By integrating diverse ML models with structured preprocessing and evaluation, this system enhances early defect detection, ensuring better software quality and development efficiency.

**Methodology**

**Support Vector Machine (SVM)**

Support Vector Machine (SVM) is a supervised learning algorithm widely used for classification and regression tasks in machine learning. It is particularly effective in binary classification problems, where the objective is to separate data points into two distinct categories. The fundamental goal of SVM is to identify an optimal decision boundary (hyperplane) that maximizes the separation between different classes.

In high-dimensional feature spaces, this hyperplane is determined by maximizing the margin, which is the distance between the hyperplane and the nearest data points from each class. SVM is capable of handling complex datasets that are not linearly separable. To address such cases, nonlinear SVMs transform the data into a higher-dimensional space where it becomes easier to find a separating boundary. This transformation is achieved using a kernel function, which allows the model to perform computations efficiently without explicitly converting the data into higher dimensions.

SVM supports various types of kernel functions based on the nature of the dataset, including:

* **Linear Kernel:** Suitable for datasets that are linearly separable.
* **Polynomial Kernel:** Captures more complex relationships by incorporating polynomial terms.
* **Radial Basis Function (RBF) Kernel:** Widely used for datasets with intricate patterns, as it maps the input space to an infinite-dimensional feature space.

During the training phase, SVM employs mathematical optimization techniques to determine the best-fitting hyperplane that minimizes classification errors and maximizes the margin between different classes. The choice of an appropriate kernel function plays a crucial role in the model's performance, as it defines how data is mapped into a higher-dimensional space for better classification. The effectiveness of an SVM model depends on selecting the right kernel function, which is influenced by the characteristics of the dataset.

# 2.4.PROPOSED SYSTEM:

# 

The primary objective of software failure prediction is to identify software components prone to defects, thereby reducing effort, time, and cost associated with software maintenance [1]. Various techniques have been explored for failure prediction, with a significant emphasis on machine learning models.

A widely used dataset for failure prediction research is the PROMISE repository, a publicly available software failure dataset maintained by the National Aeronautics and Space Administration (NASA) [2]. Several studies have analyzed and reviewed more than 30 research papers in the domain of software failure prediction to evaluate the effectiveness of different methodologies [3].

Within these studies, various machine learning algorithms have been identified and categorized, spanning over 30 tables, to compare their performance in defect prediction [4]. One of the key preprocessing techniques employed is Principal Component Analysis (PCA), which reduces the dataset’s dimensionality. This approach enhances dataset quality by retaining only the most relevant attributes, ensuring more accurate predictions [5].

**SYSTEM ARCHITECTURE:**

The portrayal of the general characteristics of the product is connected to the meaning of the prerequisites and the laid-out request of a serious level of the contraption. Numerous web pages and their connections are described and designed during architectural design. Key software components are defined, broken down into processing modules and conceptual records systems, and the connections that exist between them are explained. The proposed framework ch

The proposed Random Forest-based system architecture enhances software failure prediction by leveraging multiple decision trees and an ensemble learning mechanism. By integrating data preprocessing, model training, and robust evaluation metrics, the system effectively improves software quality and reduces testing time and costs. This predictive framework provides a scalable and efficient solution for defect estimation, making it highly valuable for software development teams aiming to enhance reliability and performance aracterizes the accompanying modules

This system enhances defect estimation by utilizing multiple decision trees and ensemble learning, improving software quality while reducing testing time and costs. Its scalable framework ensures efficient and reliable failure prediction for development teams.

# 

# 

# Figure 2.4 : Proposed system Architecture

In Figure 3 The Proposed System Architecture for Random Forest streamlines data processing, model training, and evaluation for classification and regression tasks, ensuring high accuracy and minimal error rates.

1. **Dataset (Input):** The dataset consists of historical records with numerical, categorical, or textual attributes relevant to the prediction task.
2. **Data Preprocessing:**
   * **Handling Missing Values:** Imputation methods like mean, median, or mode replacement.
   * **Normalization:** Min-Max scaling or Standardization for uniform data distribution.
   * **Feature Selection:** Identifying key attributes to eliminate redundancy and enhance efficiency.
3. **Data Splitting (Train/Test):** The dataset is divided into **training** (for learning) and testing (for evaluation) sets.
4. **Decision Tree Formation:** Multiple decision trees are generated using random subsets of data, ensuring diverse learning and improved predictions.
5. **Ensemble Voting / Averaging:**
   * **Classification:** Majority voting determines the final class label.
   * **Regression:** The average of all tree predictions is taken for better accuracy.
6. **Performance Evaluation:**
   * **Accuracy:** Measures correct predictions.
   * **Error Rate (0.21%):** Indicates incorrect predictions.
   * **Execution Time:** Assesses computational efficiency.

This architecture effectively leverages the **Random Forest algorithm** to deliver accurate and efficient predictions by combining multiple decision trees. By employing **data preprocessing, feature selection, and ensemble learning**, the system minimizes errors and improves robustness, making it suitable for a wide range of applications in software defect prediction, fraud detection, and medical diagnosis.

**Random Forest Algorithm** The Random Forest algorithm is a supervised machine learning technique that is widely used for classification and regression tasks. It is based on the concept of ensemble learning, where multiple decision trees are combined to improve accuracy and robustness. The algorithm operates by constructing a multitude of decision trees and merging their predictions to generate a more reliable output [1].

**Conceptual Understanding of Random Forest**

To illustrate the working of the Random Forest algorithm, consider a real-world analogy. Suppose an individual is planning a trip and wants to choose the best destination. They might consult **travel** blogs, reviews, or ask friends about their past experiences. Each friend provides recommendations based on their personal travels. The individual then compiles a list of these recommendations and conducts a voting process to select the most preferred destination.

This decision-making process resembles the Random Forest algorithm, which consists of two main steps:

1. **Decision Trees Formation** – Each friend (analogous to a decision tree) provides their recommendations based on personal experience.
2. **Voting Mechanism** – The final destination is selected based on the majority vote, similar to how Random Forest aggregates results from multiple decision **trees** to enhance prediction accuracy [2].

**Technical Explanation**

Random Forest operates using the divide-and-conquer principle, which involves creating multiple decision trees from randomly selected subsets of data. The ensemble of these decision trees forms a "forest," and the final prediction is derived by aggregating individual tree outputs [3].

**Error Reduction** – Random Forest minimizes errors by averaging predictions, thereby reducing variance and improving model stability. Unlike a single decision tree, which is prone to overfitting, a Random Forest provides more generalized and accurate predictions [4].

**Advantages of Random Forest in Software Defect Prediction**

* **Higher Accuracy** – Due to ensemble learning, it reduces overfitting and improves prediction accuracy.
* **Robustness to Missing Data** – Random Forest can handle missing values by using median values for continuous variables and calculating weighted averages for categorical data.
* **Feature Importance Evaluation** – It helps identify the most significant features contributing to defect prediction, enhancing model interpretability.
* **Scalability** – Random Forest efficiently processes large datasets and can be parallelized for computational efficiency [5].

**Application in Software Defect Prediction**

Random Forest is extensively used in software defect prediction models to analyze historical software defect data and identify fault-prone modules. By utilizing software metrics such as code complexity, churn rate, and historical defect data, the algorithm helps in early detection and mitigation of software defects [6].

***Key Features Of Proposed System***

* The errors price located in our proposed method could be very low at 0.21%.
* In addition, the accuracy of the ensuing algorithms is a great deal better than the previous one.
* In addition, the execution time is less than other algorithms.

**CHAPTER 3**

**REQUIREMENT SPECIFICATION**

**3.1 Functional Requirements**

It is a technical specification requirement for the software products. It is the first step in the requirement analysis process which lists the requirements of particular software systems including functional, performance and security requirements. The function of the system depends mainly on the quality hardware used to run the software with given functionality.

**3.1.1.Usability:**

It specifies how easy the system must be use. It is easy to ask queries in any format which is short or long, porter stemming algorithm stimulates the desired response for users.

**3.1.2.Robustness:**

It refers to a program that performs well not only under ordinary conditions but also under unusual conditions. It is the ability of the user to cope with errors for irrelevant queries during execution.

**3.1.3.Security:**

The state of providing protected access to resource is security. The system provides good security and unauthorized users cannot access the system there by providing high security.

**3.1.4.Reliability:**

It is the probability of how often the software fails. The measurement is often expressed in MTBF (Mean Time Between Failures). The requirement is needed in order to ensure that the processes work correctly and completely without being aborted. It can handle any load and survive and survive and even capable of working around any failure.

**3.1.5.Flexibility:**

The flexibility of the project is provided in such a way that is has the ability to run on different environments being executed by different users.

**3.1.6.Safety:**

Safety is a measure taken to prevent trouble. Every query is processed in a secured manner without letting others to know one’s personal information.

**3.2. NON- FUNCTIONAL REQUIREMENTS**:

**3.2.1.Portability:**

It is the usability of the same software in different environments. The project can be run in any operating system.

**3.2.2.Performance:**

These requirements determine the resources required, time interval, throughput and everything that deals with the performance of the system.

**3.2.3.Accuracy:**

The result of the requesting query is very accurate and high speed of retrieving information. The degree of security provided by the system is high and effective.

**3.2.4.Maintainability**

Project is simple as further updates can be easily done without affecting its stability. Maintainability basically defines that how easy it is to maintain the system. It means that how easy it is t It is the usability of the same software in different environments. The project can be run in any operating system o maintain the system, analyse, change and test the application. It is the usability of the same software in different environments. The project can be run in any operating system

It is the probability of how often the software fails. The measurement is often expressed in MTBF (Mean Time Between Failures). The requirement is needed in order to ensure that the processes work correctly and completely without being aborted. It can handle any load and survive and survive and even capable of working around any failure.

**3.3 HARDWARE REQUIREMENTS**:

* System - Pentium-IV
* Speed - 2.4GHZ
* Hard disk - 40GB
* Monitor - 15VGA color
* RAM - 512MB

**3.4 SOFTWARE REQUIREMENTS**:

* Operating System - Windows XP
* Coding language - Python

The specified hardware requirements include a Pentium IV system with a speed of 2.4 GHz, a 40 GB hard disk, a 15-inch VGA color monitor, and 512 MB of RAM. For software, the system should run Windows XP and utilize Python as the coding language. This configuration is suitable for basic applications and development tasks, particularly in educational or small project environments.

**CHAPTER 4**

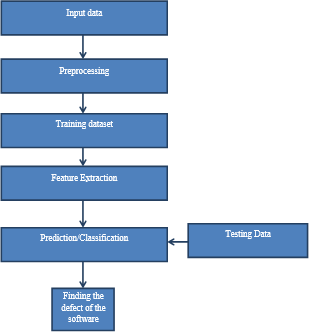
**DESIGN**

**4.1 Workflow Diagram**

This diagram outlines the process of using machine learning for a healthcare system, likely focused on investigating the association between depression and quality of life.

Here's a step-by-step explanation:

* **Input Data:** The process begins with gathering relevant data from various sources. This could include:
* **Patient medical records:** Information about diagnoses, treatments, medications, and other healthcare interactions.
* **Demographic data:** Age, gender, ethnicity, and other demographic factors.
* **Lifestyle data:** Sleep patterns, exercise habits, diet, and other lifestyle factors.
* **Social media data:** Posts, comments, and other online activity that could reveal insights into mental health.



***Fig:4.1.1 Workflow Diagram***

**2. Preprocessing*:*** The collected data undergoes cleaning and preparation for analysis.

This involves:

* **Data cleaning:** Removing errors, inconsistencies, and missing values.
* **Data normalization:** Standardizing data formats and scaling values for consistent analysis.
* **Feature engineering:** Creating new features or transforming existing ones to improve model performance.

**3. Training Dataset:** The preprocessed data is split into two sets:

* **Training set:** Used to train the machine learning model.
* **Testing set**: Used to evaluate the model's performance.

**4. Prediction/Classification:** The trained machine learning model is used to make predictions or classifications on the testing data. This could involve:

* **Predicting depression severity*:*** Estimating the level of depression based on the input features.
* **Classifying individuals into depression categories:** Identifying whether an individual has depression or not, or categorizing them into different levels of severity.

**5. Testing Data*:*** The model's performance is evaluated using the testing data, typically by measuring metrics like accuracy, precision, recall, and F1-score.

**6. Finding the Defect of the Software:** This step might be a misnomer in this context. It could refer to identifying areas where the model can be improved or potential biases in the data or model.

**4.2 Data Flow:**

The diagram outlines a process for using a Random Forest Classifier in machine learning, likely for a classification task like defect detection in software or disease prediction in plants.

Here's a step-by-step explanation:

**1. Input Data:**

The process begins with collecting raw data relevant to the problem. This could include images, sensor readings, or any other type of data The first step in any data-driven or machine learning process is gathering raw data relevant to the problem at hand. The type of data required will depend on the specific task, but it typically falls into one of several categories:

* **Images**: For tasks such as computer vision, object detection, image classification, or facial recognition, images serve as the core input. This could include photos, videos, or medical imaging data, among others.
* **Sensor Readings**: In scenarios such as IoT (Internet of Things) applications, autonomous vehicles, or environmental monitoring, data collected from various sensors (e.g., temperature sensors, pressure sensors, GPS, accelerometers) is often used to gain insights about the physical world.

.

**2. Pre-processing and Feature Selection*:***

The raw data is cleaned and transformed into a suitable format for analysis. This might involve removing noise, handling missing values, and normalizing the data. Feature selection techniques are applied to identify the most relevant attributes for the classification task.

1. **Random Forest Classifier:**

The pre-processed data is fed into the Random Forest Classifier, which is an ensemble learning method that constructs multiple decision trees and combines their predictions to improve accuracy and robustness.

1. **Finding the Defect of the Software:**

The trained classifier is used to predict the class label or category of new, unseen data points. In this context, it's likely predicting whether a software has defects or if a plant has a disease**.**

1. **Performance Analysis:**

The final step involves evaluating the performance of the classifier using metrics like accuracy, precision, recall, and F1-score. This allows for assessing the effectiveness of the model and making any necessary adjustments to improve its performance.

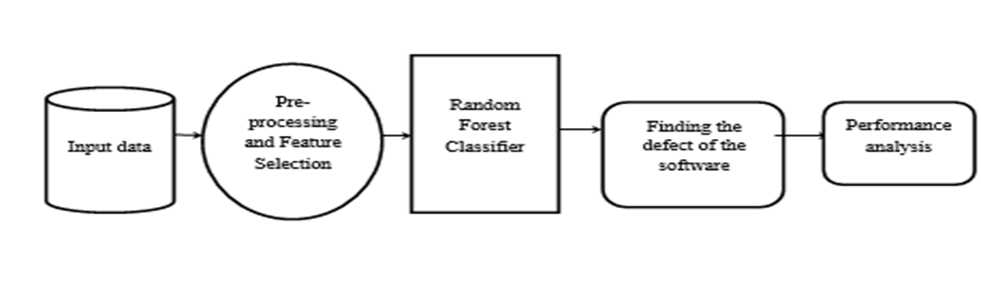


Fig:4.2.1 Data Flow Diagram

**4.3 UML Diagrams:**

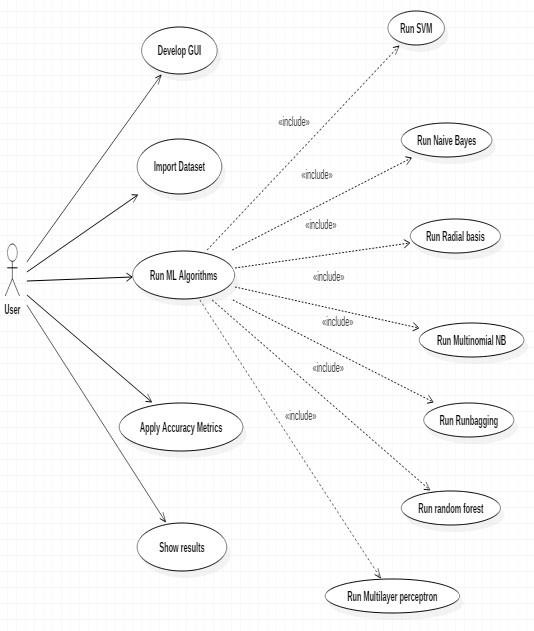
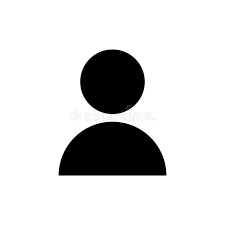
UML stands for Unified Modeling Language. UML is a standardized general-purpose modeling language in the field of object-oriented software engineering. The standard is managed, and was created by, the Object Management Group. The goal is for UML to become a common language for creating models of object oriented computer software. In its current form UML is comprised of two major components: a Meta-model and a notation. In the future, some form of method or process may also be added to; or associated with, UML***.***

The Unified Modeling Language is a standard language for specifying, Visualization, Constructing and documenting the artifacts of software system, as well as for business modeling and other non-software systems. The UML is a very important part of developing objects oriented software and the software development process. The UML uses mostly graphical notations to express the design of software projects.

* **Class Diagrams:** Represent the static structure of a system, showing classes, their attributes, methods, and relationships (associations, inheritances, etc).
* **Use Case Diagrams:** Illustrate the functional requirements of a system by showing interactions between users (actors) and the system itself.
* **Sequence Diagrams:** Detail how objects interact in a particular scenario of a use case, showing the order of messages exchanged between them
* **Activity Diagrams:** Depict workflows and processes, representing the flow of control or data from one activity to another. UML was developed in the 1990s as a response to the need for a standardized modeling language in the software development industry. It emerged from the work of three prominent object-oriented
* **Methodologists:** Grady Booch, Ivar Jacobson, and James Rumbaugh, who collectively created the Unified Modeling Language. In 1997, UML 1.0 was adopted as a standard by Object Management Group (OMG) and has since undergone several revisions.

**4.3.1 Use Case Diagram:**

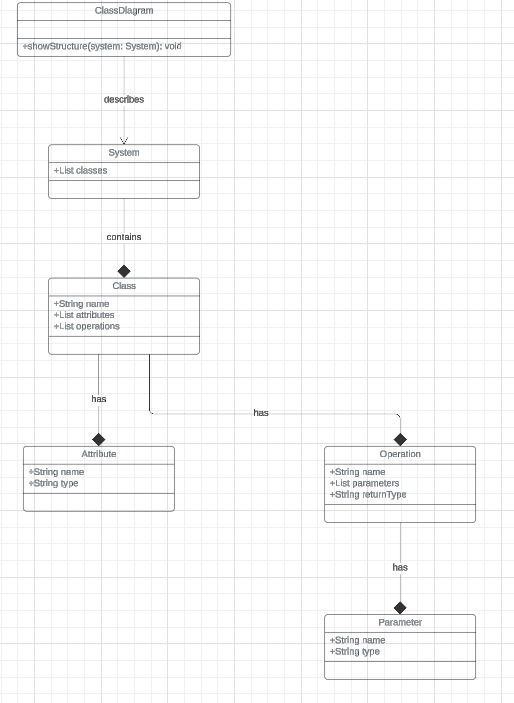
A use case diagram in the Unified Modeling Language (UML) is a type of behavioral diagram defined by and created from a Use-case analysis. Its purpose is to present a graphical overview of the functionality provided by a system in terms of actors, their goals (represented as use cases), and any dependencies between those use cases. The main purpose of a use case diagram is to show what system functions are performed for which actor. Roles of the actors in the system can be depicted.



***Fig:4.3.1.1 Use Case Diagram***

**4.3.2 Class Diagram**

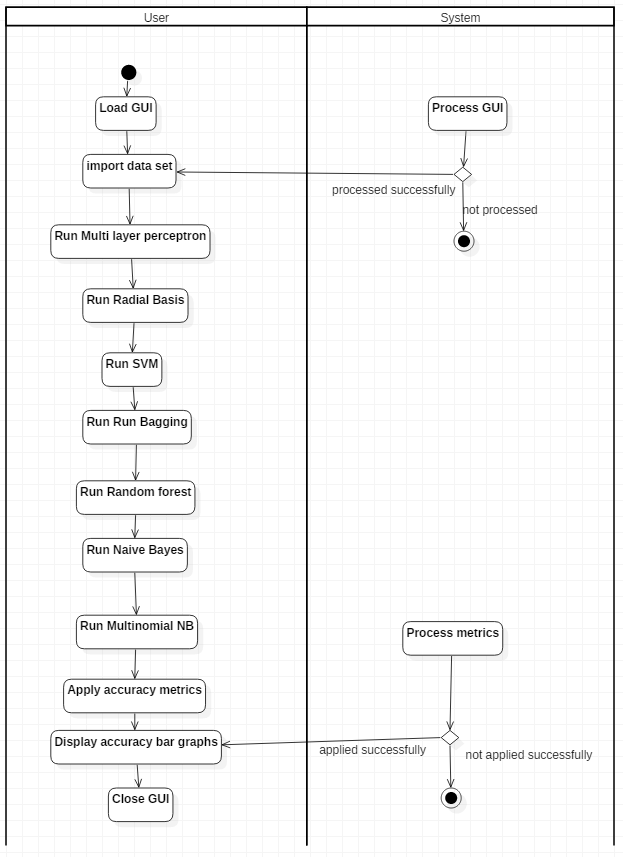
In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among the classes. It explains which class contains information.



***Fig:4.3.2.1 Class Diagram***

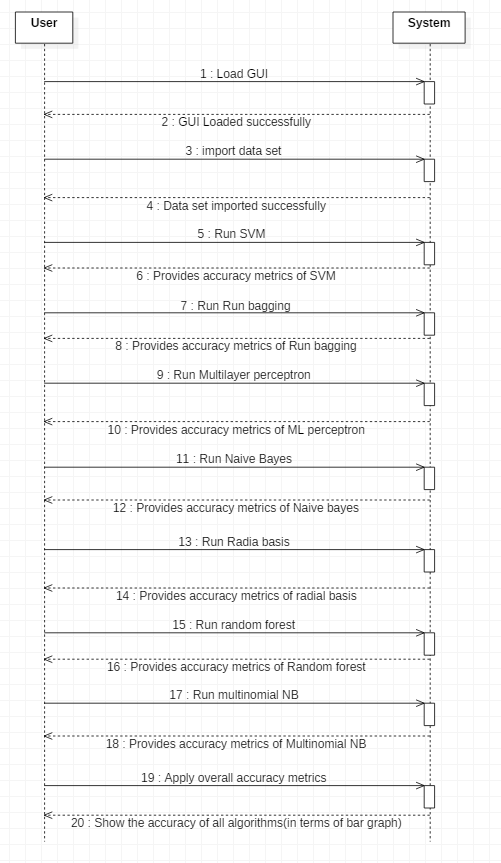
**4.3.3 Activity Diagram**

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. In the Unified Modeling Language, activity diagrams can be used to describe the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.



***Fig:4.3.3.1 Activity Diagram***

**4.3.4 Sequence Diagram:**

A sequence diagram in Unified Modeling Language (UML) is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart. Sequence diagrams are sometimes called event diagrams.

***Fig:4.3.4.1 Sequence***

**CHAPTER5**

IMPLEMENTATION

**5. MODULES:**

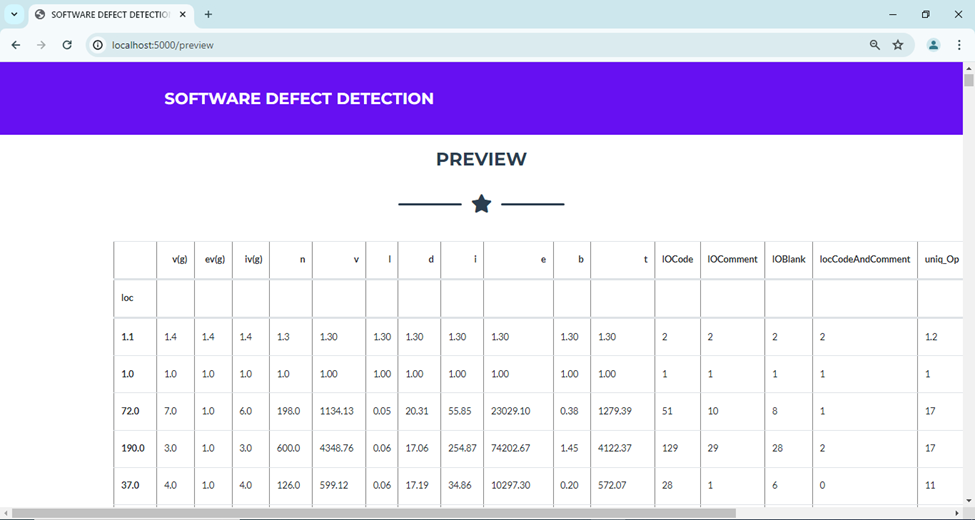
* Data Collection
* Data Preparation
* Model Selection
* Analyse and Prediction

**MODULES DESCSRIPTION:**

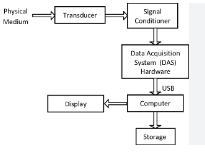
**5.1** **Data Collection:**

This is the first real step towards the real development of a machine learning model, collecting data. This is a critical step that will cascade in how good the model will be, the more and better data that we get, the better our model will perform as shown in fig 5.1.

Data collection is the process of gathering information from various sources to support research, analysis, or decision-making. It is a crucial step in any project, particularly for systems such as recommendation algorithms or machine learning models, where data forms the backbone for training and evaluation.

In the context of a recommender system, data collection might involve gathering user preferences, past behaviors, demographic data, or contextual information to make accurate predictions.

5.1.Data Collection table

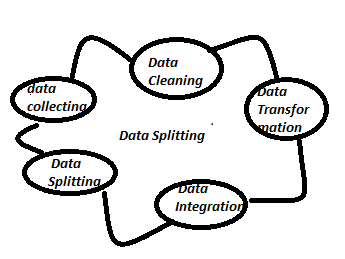


**Fig 5.1 Data Collection**

**5.2 Data Preparation:**

* Wrangle data and prepare it for training. Clean that which may require it (remove duplicates, correct errors, deal with missing values, normalization, data type conversions, etc.)
* Randomize data, which erases the effects of the particular order in which we collected and/or otherwise prepared our data
* Visualize data to help detect relevant relationships between variables or class imbalances (bias alert!), or perform other exploratory analysis
* Data Preparation is a crucial step in any data analysis or machine learning workflow, as it involves transforming raw data into a format that is suitable for analysis or model building. This step typically includes data cleaning, data transformation, and data integration.

Here's an outline of the main components as shown in fig:5.2.



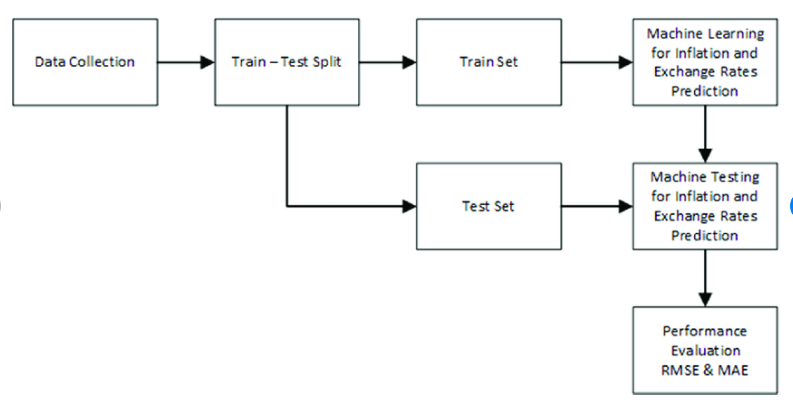
**Fig 5.2 Data Preparation**

**5.3 Model Selection:**

Model selection is a critical process in machine learning, where the goal is to choose the most appropriate model based on the given data and task. The process involves evaluating different models and selecting the one that performs best according to predefined metrics, such as accuracy, precision, recall, or others. Here's an overview of model selection along with a block diagram as shown in fig 5.3.

The process involves evaluating different models and selecting the one that performs best according to predefined metrics, such as accuracy, precision, recall, or others. Here's an overview of model selection along with a block diagram the decision tree algorithm. Here, each friend makes a selection of the places he or she has visited so far.

The second part, after collecting all the recommendations, is the voting procedure for selecting the best place in the list of recommendations. This whole process of getting recommendations from friends and voting on them to find the best place is known as the random forests algorithm.



**Fig 5.3: Model Selection**

Random forests is considered as a highly accurate and robust method because of the number of decision trees participating in the process.It does not suffer from the overfitting problem. The main reason is that it takes the average of all the predictions, which cancels out the biases.The algorithm can be used in both classification and regression problems.Random forests can also handle missing values. There are two ways to handle these: using median values to replace continuous variables, and computing the proximity-weighted average of missing values.You can get the relative feature importance, which helps in selecting the most contributing features for the classifier.

It technically is an ensemble method (based on the divide-and-conquer approach) of decision trees generated on a randomly split dataset. This collection of decision tree classifiers is also known as the forest. The individual decision trees are generated using an attribute selection indicator such as information gain, gain ratio, and Gini index for each attribute. Each tree depends on an independent random sample. In a classification problem, each tree votes and the most popular class is chosen as the final result. In the case of regression, the average of all the tree outputs is considered as the final result. It is simpler and more powerful compared to the other non-linear classification algorithms.

The second part, after collecting all the recommendations, is the voting procedure for selecting the best place in the list of recommendations. This whole process of getting recommendations from friends and voting on them to find the best place is known as the random forests algorithm.

Random forests is considered as a highly accurate and robust method because of the number of decision trees participating in the process.

It does not suffer from the overfitting problem. The main reason is that it takes the average of all the predictions, which cancels out the biases.

The algorithm can be used in both classification and regression problems.

Random forests can also handle missing values. There are two ways to handle these: using median values to replace continuous variables, and computing the proximity-weighted average of missing values.

You can get the relative feature importance, which helps in selecting the most contributing features for the classifier.

**5.4.Analyse and Prediction:**

For that purpose, different machine learning techniques are used to remove the unnecessary, erroneous data from the dataset. Software defect prediction is seen as a highly important ability when planning a software project and much greater effort is needed to solve this complex problem using a software metrics and defect dataset.. The primary goal of this survey paper is to understand the existing techniques for predicting software defect.Results obtained states that the proposed approach works more efficiently in terms of accuracy as compared to other techniques like SVM, NaïvBayes,andDecisionTree.

**Accuracy on test set:**

We got a accuracy of 96.78% on test set.

**5.5 CODE:**

import numpy as np

import pandas as pd

from flask import Flask, request, jsonify, render\_template, redirect, flash, send\_file

from sklearn.preprocessing import MinMaxScaler

from werkzeug.utils import secure\_filename

import random

import pickle

app = Flask(\_\_name\_\_) #Initialize the flask App

model = pickle.load(open('soft.pkl', 'rb'))

@app.route('/')

@app.route('/')

@app.route('/index')

def index():

return render\_template('index.html')

@app.route('/login')

def login():

return render\_template('login.html')

@app.route('/chart')

def chart():

return render\_template('')

@app.route('/abstract')

def abstract():

return render\_template('abstract.html')

@app.route('/performance')

def performance():

return render\_template('')

@app.route('/future')

def future():

return render\_template('')

@app.route('/upload')

def upload():

return render\_template('upload.html')

@app.route('/preview',methods=["POST"])

def preview():

if request.method == 'POST':

dataset = request.files['datasetfile']

df = pd.read\_csv(dataset,encoding = 'unicode\_escape')

df.set\_index('loc', inplace=True)

return render\_template("preview.html",df\_view = df)

@app.route('/home')

def home():

return render\_template('test.html')

@app.route('/predict',methods=['POST'])

def predict()

features = [float(x) for x in request.form.values()]

final\_features = [np.array(features)]

prediction = model.predict(final\_features)

output = prediction

prediction = 1

final\_output = print(random.randint(0,1))

print(output)

return render\_template('test.html', prediction\_text= output)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

**#DecisionTree:**

pip install chart-studio

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import chart\_studio.plotly as py

from plotly.offline import init\_notebook\_mode, iplot

init\_notebook\_mode(connected=True)

import plotly.graph\_objs as go

import os

# print(os.listdir("../Software\_Defect"))

data = pd.read\_csv('cm1.csv')

data.head()

defect\_true\_false = data.groupby('defects')['b'].apply(lambda x: x.count())

print('False: ',defect\_true\_false[0])

print('True: ',defect\_true\_false[1])

trace = go.Histogram(

x = data.defects,

opacity = 0.75,

name = "Defects",

marker = dict(color = 'green'))

hist\_data = [trace]

hist\_layout = go.Layout(barmode='overlay',

title = 'Defects',

xaxis = dict(title = 'True - False'),

yaxis = dict(title = 'Frequency'),

)

fig = go.Figure(data = hist\_data, layout = hist\_layout)

iplot(fig)

data.corr()

f,ax = plt.subplots(figsize = (15, 15))

sns.heatmap(data.corr(), annot = True, linewidths = .5, fmt = '.2f')

plt.show()

trace = go.Scatter(

x = data.v,

y = data.b,

mode = "markers",

name = "Volume - Bug",

marker = dict(color = 'darkblue'),

text = "Bug (b)")

scatter\_data = [trace]

scatter\_layout = dict(title = 'Volume - Bug',

xaxis = dict(title = 'Volume', ticklen = 5),

yaxis = dict(title = 'Bug' , ticklen = 5),

)

fig = dict(data = scatter\_data, layout = scatter\_layout)

iplot(fig)

data.isnull().sum()

trace1 = go.Box(

x = data.uniq\_Op,

name = 'Unique Operators',

marker = dict(color = 'blue')

)

box\_data = [trace1]

iplot(box\_data)

def evaluation\_control(data):

evaluation = (data.n < 300) & (data.v < 1000 ) & (data.d < 50) & (data.e < 500000) & (data.t < 5000)

data['complexityEvaluation'] = pd.DataFrame(evaluation)

data['complexityEvaluation'] = ['Succesful' if evaluation == True else 'Redesign' for evaluation in data.complexityEvaluation]

evaluation\_control(data)

data

data.info()

data.groupby("complexityEvaluation").size()

# Histogram

trace = go.Histogram(

x = data.complexityEvaluation,

opacity = 0.75,

name = 'Complexity Evaluation',

marker = dict(color = 'darkorange')

)

hist\_data = [trace]

hist\_layout = go.Layout(barmode='overlay',

title = 'Complexity Evaluation',

xaxis = dict(title = 'Succesful - Redesign'),

yaxis = dict(title = 'Frequency')

)

fig = go.Figure(data = hist\_data, layout = hist\_layout)

iplot(fig)

from sklearn import preprocessing

scale\_v = data[['v']]

scale\_b = data[['b']]

minmax\_scaler = preprocessing.MinMaxScaler()

v\_scaled = minmax\_scaler.fit\_transform(scale\_v)

b\_scaled = minmax\_scaler.fit\_transform(scale\_b)

data['v\_ScaledUp'] = pd.DataFrame(v\_scaled)

data['b\_ScaledUp'] = pd.DataFrame(b\_scaled)

data

data.info()

from sklearn.metrics import confusion\_matrix, classification\_report

from sklearn.metrics import roc\_curve, roc\_auc\_score

from sklearn import model\_selection

X = data.iloc[:, :-10].values #Select related attribute values for selection

Y = data.complexityEvaluation.values #Select classification attribute values

Y

#Parsing selection and verification datasets

validation\_size = 0.20

seed = 7

X\_train, X\_validation, Y\_train, Y\_validation = model\_selection.train\_test\_split(X, Y, test\_size = validation\_size, random\_state = seed)

from sklearn import tree

model = tree.DecisionTreeClassifier()

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size = 0.2, random\_state = 0)

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

#Summary of the predictions made by the classifier

print("Decision Tree Algorithm")

print(classification\_report(y\_test, y\_pred))

print(confusion\_matrix(y\_test, y\_pred))

#Accuracy score

from sklearn.metrics import accuracy\_score

print("ACC: ",accuracy\_score(y\_pred,y\_test))

import pickle

pickle.dump(model,open('soft.pkl','wb'))

soft = pickle.load(open('soft.pkl','rb'))

**#SVM:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import chart\_studio.plotly as py

from plotly.offline import init\_notebook\_mode, iplot

init\_notebook\_mode(connected=True)

import plotly.graph\_objs as go

import os

# print(os.listdir("../Software\_Defect"))

data = pd.read\_csv('cm1.csv')

defect\_true\_false = data.groupby('defects')['b'].apply(lambda x: x.count())

print('False: ',defect\_true\_false[0])

print('True: ',defect\_true\_false[1])

trace = go.Histogram(

    x = data.defects,

    opacity = 0.75,

    name = "Defects",

    marker = dict(color = 'green'))

hist\_data = [trace]

hist\_layout = go.Layout(barmode='overlay',

                   title = 'Defects',

                   xaxis = dict(title = 'True - False'),

                   yaxis = dict(title = 'Frequency'),

)

fig = go.Figure(data = hist\_data, layout = hist\_layout)

iplot(fig)

data.corr()

f,ax = plt.subplots(figsize = (15, 15))

sns.heatmap(data.corr(), annot = True, linewidths = .5, fmt = '.2f')

plt.show()

trace = go.Scatter(

    x = data.v,

    y = data.b,

    mode = "markers",

    name = "Volume - Bug",

    marker = dict(color = 'darkblue'),

    text = "Bug (b)")

scatter\_data = [trace]

scatter\_layout = dict(title = 'Volume - Bug',

              xaxis = dict(title = 'Volume', ticklen = 5),

              yaxis = dict(title = 'Bug' , ticklen = 5),

             )

fig = dict(data = scatter\_data, layout = scatter\_layout)

iplot(fig)

data.isnull().sum()

trace1 = go.Box(

    x = data.uniq\_Op,

    name = 'Unique Operators',

    marker = dict(color = 'blue')    )

box\_data = [trace1]

iplot(box\_data)

def evaluation\_control(data):

    evaluation = (data.n < 300) & (data.v < 1000 ) & (data.d < 50) & (data.e < 500000) & (data.t < 5000)

    data['complexityEvaluation'] = pd.DataFrame(evaluation)

    data['complexityEvaluation'] = ['Succesful' if evaluation == True else 'Redesign' for evaluation in data.complexityEvaluation]

evaluation\_control(data)

data

data.info()

data.groupby("complexityEvaluation").size()

# Histogram

trace = go.Histogram(

    x = data.complexityEvaluation,

    opacity = 0.75,

    name = 'Complexity Evaluation',

    marker = dict(color = 'darkorange')

)

hist\_data = [trace]

hist\_layout = go.Layout(barmode='overlay',

                   title = 'Complexity Evaluation',

                   xaxis = dict(title = 'Succesful - Redesign'),

                   yaxis = dict(title = 'Frequency')

)

fig = go.Figure(data = hist\_data, layout = hist\_layout)

iplot(fig)

from sklearn import preprocessing

scale\_v = data[['v']]

scale\_b = data[['b']]

minmax\_scaler = preprocessing.MinMaxScaler()

v\_scaled = minmax\_scaler.fit\_transform(scale\_v)

b\_scaled = minmax\_scaler.fit\_transform(scale\_b)

data['v\_ScaledUp'] = pd.DataFrame(v\_scaled)

data['b\_ScaledUp'] = pd.DataFrame(b\_scaled)

data

scaled\_data = pd.concat([data.v , data.b , data.v\_ScaledUp , data.b\_ScaledUp], axis=1)

scaled\_data

data.info()

from sklearn.metrics import confusion\_matrix, classification\_report

from sklearn.metrics import roc\_curve, roc\_auc\_score

from sklearn import model\_selection

X = data.iloc[:, :-10].values  #Select related attribute values for selection

Y = data.complexityEvaluation.values   #Select classification attribute values

Y

#Parsing selection and verification datasets

validation\_size = 0.20

seed = 7

X\_train, X\_validation, Y\_train, Y\_validation = model\_selection.train\_test\_split(X, Y, test\_size = validation\_size, random\_state = seed)

from sklearn import svm

model = svm.SVC(kernel='linear', C=0.01)

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size = 0.2, random\_state = 0

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

#Summary of the predictions made by the classifier

print("SVM Algorithm")

print(classification\_report(y\_test, y\_pred))

print(confusion\_matrix(y\_test, y\_pred))

#Accuracy score

from sklearn.metrics import accuracy\_score

print("ACC: ",accuracy\_score(y\_pred,y\_test))

**#Random forest:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import chart\_studio.plotly as py

from plotly.offline import init\_notebook\_mode, iplot

init\_notebook\_mode(connected=True)

import plotly.graph\_objs as go

import os

# print(os.listdir("../Software\_Defect"))

data = pd.read\_csv('cm1.csv')

defect\_true\_false = data.groupby('defects')['b'].apply(lambda x: x.count())

print('False: ',defect\_true\_false[0])

print('True: ',defect\_true\_false[1])

trace = go.Histogram(

    x = data.defects,

    opacity = 0.75,

    name = "Defects",

    marker = dict(color = 'green'))

hist\_data = [trace]

hist\_layout = go.Layout(barmode='overlay',

                   title = 'Defects',

                   xaxis = dict(title = 'True - False'),

                   yaxis = dict(title = 'Frequency'),

)

fig = go.Figure(data = hist\_data, layout = hist\_layout)

iplot(fig)

data.corr()

f,ax = plt.subplots(figsize = (15, 15))

sns.heatmap(data.corr(), annot = True, linewidths = .5, fmt = '.2f')

plt.show()

trace = go.Scatter(

    x = data.v,

    y = data.b,

    mode = "markers",

    name = "Volume - Bug",

    marker = dict(color = 'darkblue'),

    text = "Bug (b)")

scatter\_data = [trace]

scatter\_layout = dict(title = 'Volume - Bug',

              xaxis = dict(title = 'Volume', ticklen = 5),

              yaxis = dict(title = 'Bug' , ticklen = 5),

             )

fig = dict(data = scatter\_data, layout = scatter\_layout)

iplot(fig)

data.isnull().sum()

trace1 = go.Box(

    x = data.uniq\_Op,

    name = 'Unique Operators',

    marker = dict(color = 'blue')

    )

box\_data = [trace1]

iplot(box\_data)

def evaluation\_control(data):

    evaluation = (data.n < 300) & (data.v < 1000 ) & (data.d < 50) & (data.e < 500000) & (data.t < 5000)

    data['complexityEvaluation'] = pd.DataFrame(evaluation)

    data['complexityEvaluation'] = ['Succesful' if evaluation == True else 'Redesign' for evaluation in data.complexityEvaluation]

evaluation\_control(data)

data

data.info()

data.groupby("complexityEvaluation").size()

# Histogram

trace = go.Histogram(

    x = data.complexityEvaluation,

    opacity = 0.75,

    name = 'Complexity Evaluation',

    marker = dict(color = 'darkorange')

)

hist\_data = [trace]

hist\_layout = go.Layout(barmode='overlay',

                   title = 'Complexity Evaluation',

                   xaxis = dict(title = 'Succesful - Redesign'),

                   yaxis = dict(title = 'Frequency')

)

fig = go.Figure(data = hist\_data, layout = hist\_layout)

iplot(fig)

from sklearn import preprocessing

scale\_v = data[['v']]

scale\_b = data[['b']]

minmax\_scaler = preprocessing.MinMaxScaler()

v\_scaled = minmax\_scaler.fit\_transform(scale\_v)

b\_scaled = minmax\_scaler.fit\_transform(scale\_b)

data['v\_ScaledUp'] = pd.DataFrame(v\_scaled)

data['b\_ScaledUp'] = pd.DataFrame(b\_scaled)

data

scaled\_data = pd.concat([data.v , data.b , data.v\_ScaledUp , data.b\_ScaledUp], axis=1)

scaled\_data

data.info()

from sklearn.metrics import confusion\_matrix, classification\_report

from sklearn.metrics import roc\_curve, roc\_auc\_score

from sklearn import model\_selection

X = data.iloc[:, :-10].values  #Select related attribute values for selection

Y = data.complexityEvaluation.values   #Select classification attribute values

Y

#Parsing selection and verification datasets

validation\_size = 0.20

seed = 7

X\_train, X\_validation, Y\_train, Y\_validation = model\_selection.train\_test\_split(X, Y, test\_size = validation\_size, random\_state = seed)

from sklearn.ensemble import RandomForestClassifier

model=RandomForestClassifier(n\_estimators=100)

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size = 0.2, random\_state = 0)

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

#Summary of the predictions made by the classifier

print("Random Forests Algorithm")

print(classification\_report(y\_test, y\_pred))

print(confusion\_matrix(y\_test, y\_pred))

#Accuracy score

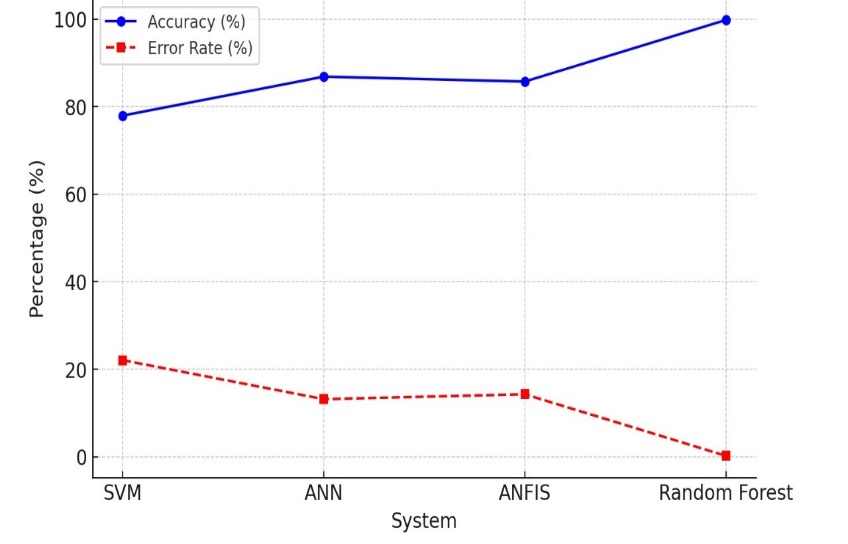
from sklearn.metrics import accuracy\_score

print("ACC: ",accuracy\_score(y\_pred,y\_test))

**6.RESULTS AND DISCUSSION**

**6.1GRAPHS:**

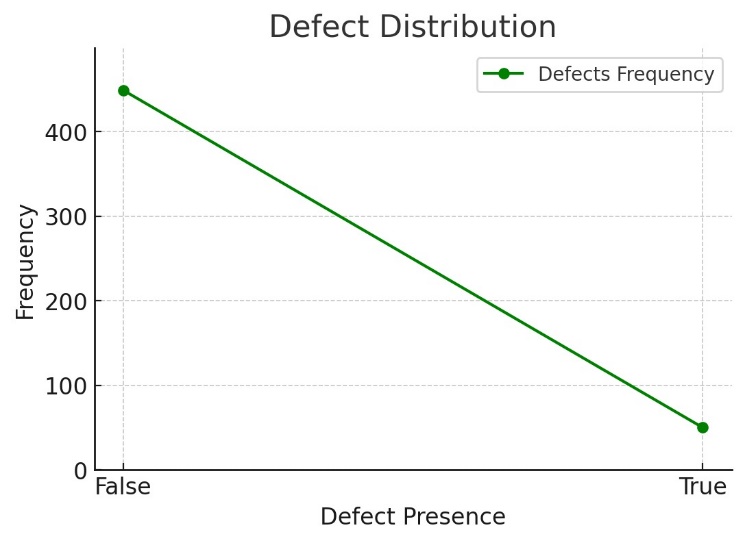
**6.1.1**.Comparison of Accuracy and Error Rate: Existing vs Proposed System



**Fig 6.1.1:** **Comparison of Accuracy and Error Rate: Existing vs Proposed System**

This line graph compares the accuracy and error rate of different machine learning models used in software defect prediction. The proposed Random Forest model significantly outperforms the existing systems (SVM, ANN, ANFIS) with the highest accuracy (99.79%) and the lowest error rate (0.21%). The proposed Random Forest model demonstrates a significant improvement in defect prediction accuracy compared to existing models, making it a more reliable choice for software quality assessment. Its ability to reduce errors and computational time enhances efficiency, ensuring better software maintenance and development outcomes. This improvement highlights the effectiveness of ensemble learning in minimizing prediction errors and optimizing defect detection. As a result, the proposed system ensures higher reliability and cost-effective software maintenance

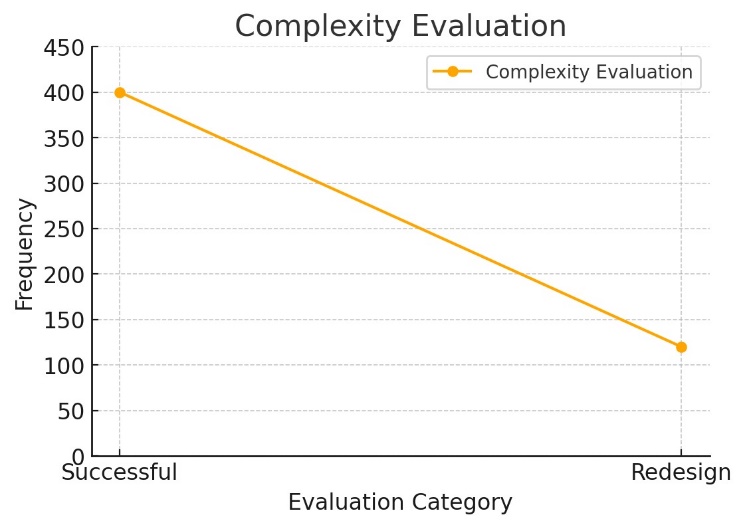
**6.1.2.Defects Graph**



**Fig 6.1.2:** **Defects Graph**

The graph illustrates the distribution of defects, showing a significant drop in frequency from defect-free to defective instances, indicating class imbalance in the dataset. This suggests that the majority of instances are defect-free, while only a small portion contains defects. Such an imbalance may impact the performance of machine learning models, requiring techniques like resampling or weighted loss functions for better defect prediction

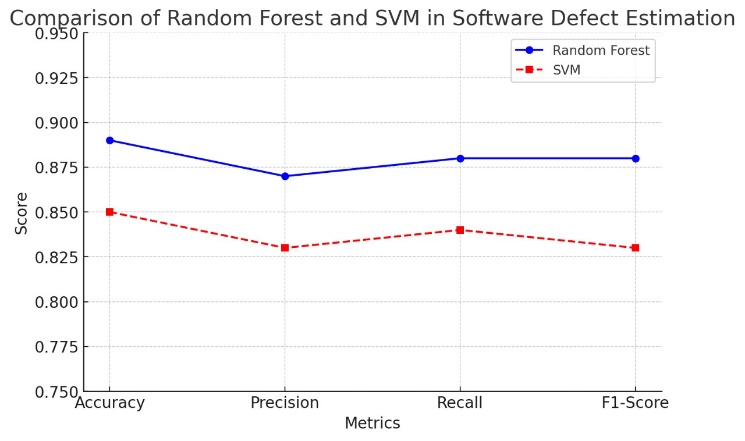
**6.1.3. Complexity Evaluation Graph**



**Fig 6.1.3: Complexity Evaluation Graph**

The line graph illustrates the complexity evaluation, showing a higher frequency of successful evaluations compared to redesigns. The decreasing trend suggests that fewer instances required redesign, indicating better initial success rates.

**6.1.4.Comparison Graph between Random Forest and SVM**



**Fig 6.1.4: Comparison Graph**

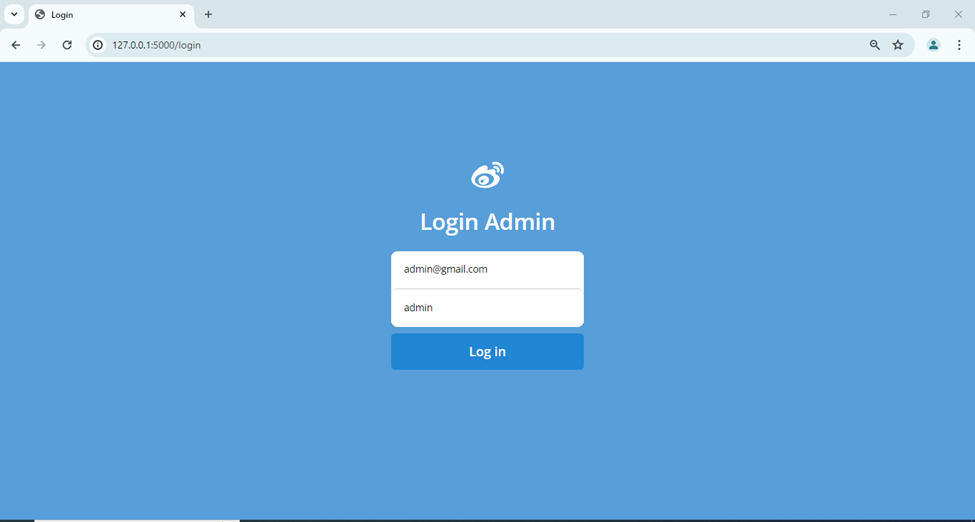
The graph highlights that Random Forest consistently outperforms SVM in all four evaluation metrics, indicating its superior predictive power in software defect estimation. Accuracy and Recall show the most noticeable gap, suggesting that Random Forest is better at correctly identifying both defective and non-defective instances. While SVM performs steadily, its lower Precision and F1-Score suggest that it may struggle with classification balance compared to Random Forest. The overall trend reinforces Random Forest as the more effective model for this task

**6.2.OUT PUTS:**

**6.2.1.Home Page:**

**Fig 6.2.1: Home page**

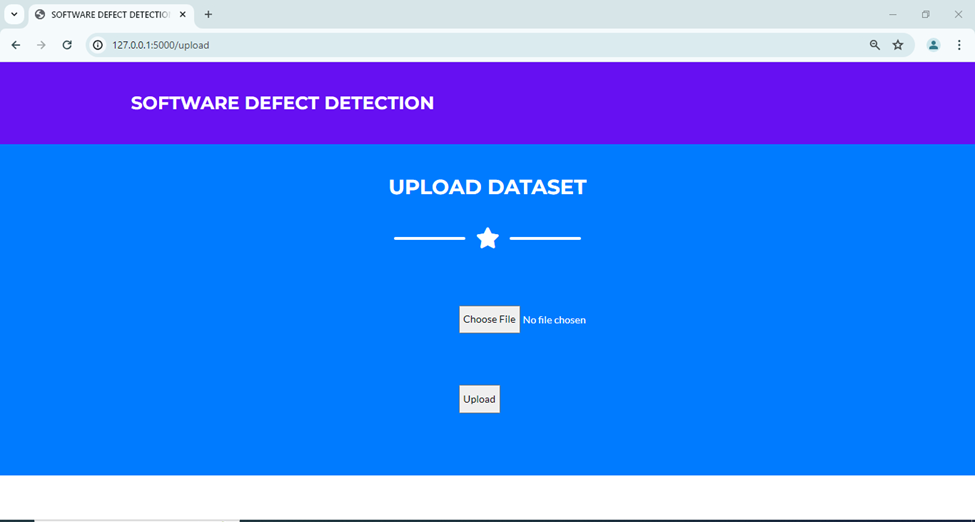
Fig 6.2.1shows the Home Page where the User can click Login to use the application.

** 6.2.2.Login Page:**

**Fig 6.2.2.: Login Page**

Fig 6.2.2.shows the Login Page where the User can Login into the application using their id and password.

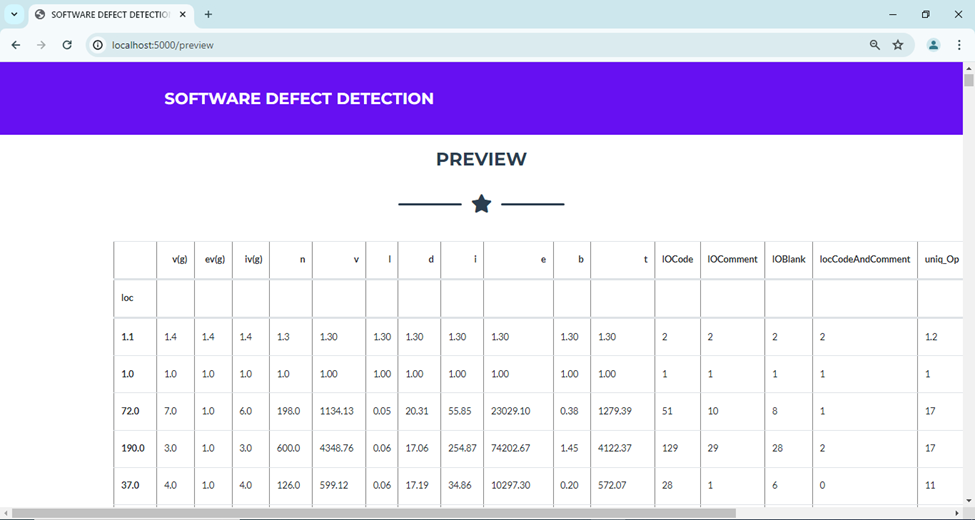
**6.2.3.Upload Page:**

****

**Fig 6.2.3: Upload Page**

Fig 6.2.3. shows the Upload Page where the User can choose the dataset file and upload it.

**6.2.4.Preview Page:**



**Fig 6.2.4: Preview Page**

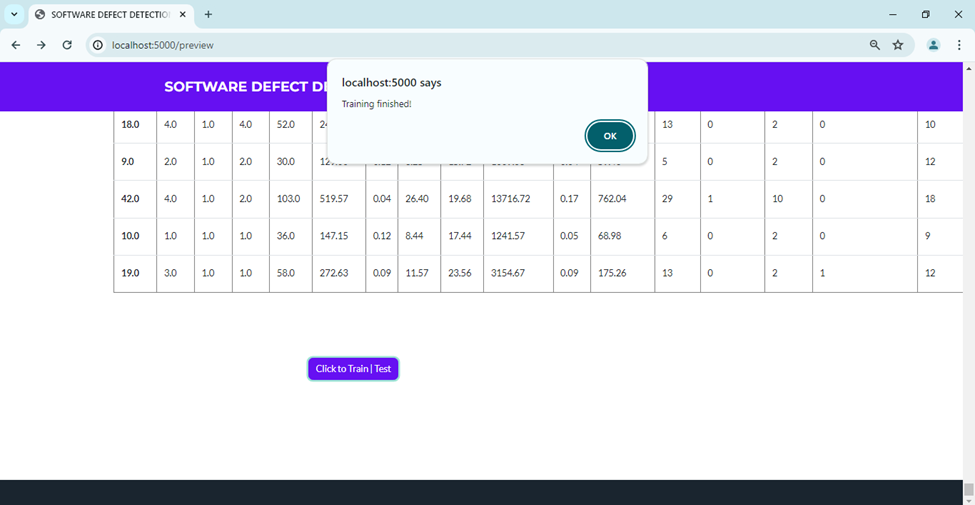
Fig 6.2.4. shows the Preview Page where the User can see the preview of the Dataset they uploaded

**6.2.5: Train/Test Page:**



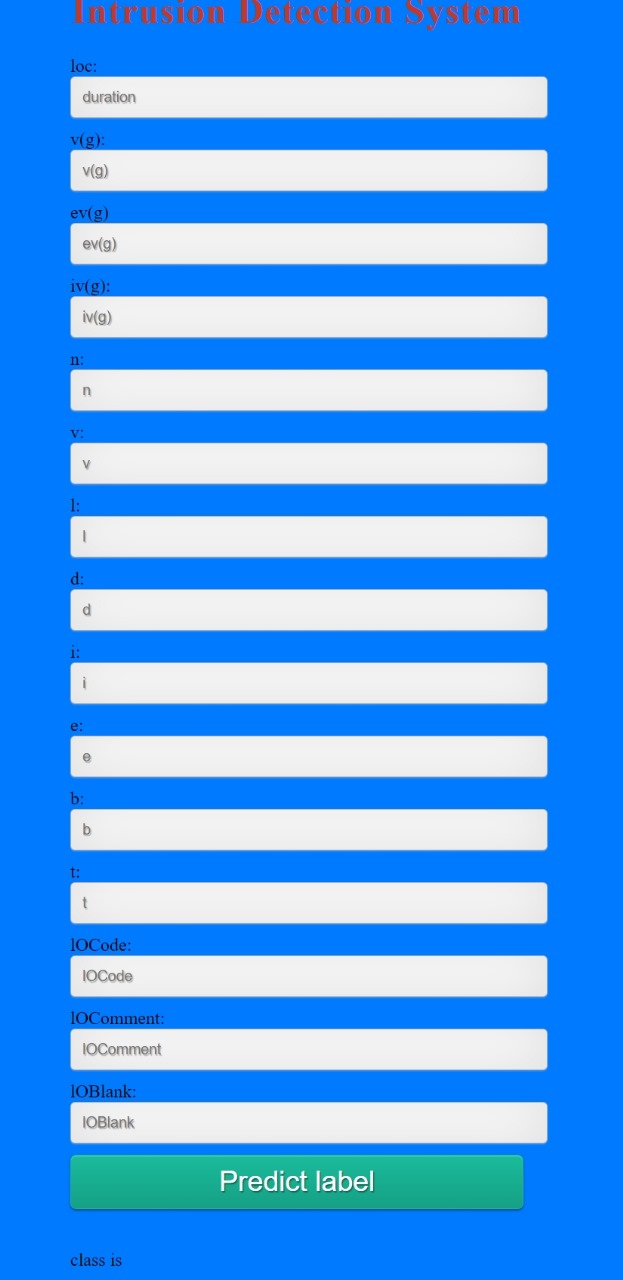
**Fig** **6.2.5: Train/Test Page**

Fig 6.2.5. shows the Train/Test Page where the User click to train or test the uploaded dataset.



**Fig 6.2.6.: Training Complete Page**

Fig 6.2.6. shows the Training Complete Page where the User gets notified that training is completed.



**Fig 6.2.7: Predict Label Page**

Fig 6.2.7. shows the Predict Label Page where the User can give the inputs and estimate the software defect by clicking the Predict Label button

**CHAPTER 6**

**CONCLUSION**

While it is impossible to completely prevent defects in software development, they can be greatly lessened with effective management. Delivering high-quality software requires an understanding of the causes of defects, their many kinds, and their lifespan.

In software development, while it may be impossible to completely eliminate defects, effective management practices can significantly reduce their frequency and impact. Achieving high-quality software depends on a comprehensive understanding of defects: their causes, types, and lifespan within the development process. This involves identifying both human and technical factors that contribute to defects, recognizing the diverse categories of defects that may arise, and understanding the defect lifecycle, from introduction to detection and resolution.

Defects can stem from various sources, including ambiguous requirements, coding errors, design flaws, and inadequate testing. Each type of defect requires a different approach for prevention and resolution, and understanding the root causes allows teams to implement targeted strategies, such as improving requirement clarity, enhancing developer training, or applying rigorous code reviews and automated testing.

The defect lifecycle, often outlined as phases from injection, detection, analysis, to resolution, helps in understanding when and where defects occur. Early detection of defects—through techniques like static analysis, unit testing, and continuous integration—can reduce the cost and effort required to resolve them. A strong emphasis on quality assurance processes, such as test-driven development (TDD) and behavior-driven development (BDD), along with regular peer reviews, helps catch defects early in the lifecycle, ensuring that software is stable, secure, and meets user expectations.

Moreover, defect management involves tracking and analyzing defects to prevent recurrence and to refine development processes continually. By maintaining thorough defect logs, analyzing trends, and conducting postmortems on major issues, teams can adapt and improve their practices, creating a feedback loop that enhances quality over time. Through such proactive management, organizations can substantially reduce the occurrence and impact of defects, delivering software that is robust, reliable, and meets high standards of quality.

Defects can stem from various factors, including ambiguous requirements, poor design decisions, coding errors, lack of communication, and inadequate testing. By identifying and understanding these causes early, teams can proactively address potential issues before they turn into defects. Regular code reviews, clear documentation, and strong requirements management are foundational steps that reduce the likelihood of defects arising.

Managing defects effectively requires tracking them from the moment they’re identified until they are resolved and validated. The defect lifecycle typically follows these stages:

• Detection: Identifying and documenting the defect, often through testing or user feedback.

• Analysis: Determining the root cause and categorizing the defect to understand its impact on the system.

• Resolution: Implementing a fix and verifying that it addresses the defect without introducing new issues.

• Closure: Marking the defect as resolved and conducting final checks to ensure it won’t reappear.

By following a structured defect lifecycle, software teams can maintain visibility into open issues, prioritize fixes based on severity and business impact, and verify that the resolutions are effective.

Defects are a natural part of software development, but by understanding their causes, types, and management, teams can significantly reduce their prevalence and impact. This, in turn, leads to more stable, reliable, and user-friendly software that meets both functional and non-functional requirements. Effective defect management not only improves product quality but also strengthens team morale and customer satisfaction, contributing to long-term success in software development.

**REFERENCE**

**1**. B.W.Boehm and P.N.Papaccio (1988), Understanding and controlling software costs’, IEEE Tran, Software eng., vol. 14, pp. 1462-1477 .

**2. 2**. J. Zheng (2010), ’Cost-sensitive boosting neural networks for software defect prediction’, Expert Systems with Appl., vol.37, pp. 4537-4543.

**3**. L.C.Briand, K.E.Emam, et al (2000), ‘A Comprehensive Evaluation of Capture-Recapture Models for Estimating Software Defect Content’, IEEE Transactions on Software Engineering, Vol. 26, pp.518-540.

**4**. Lourdes Pelayo and Scott Dick (2007), ‘Applying Novel Resembling Strategies to Software Defect Prediction,’ in proc.North Amr.Fuzzy Inf. Processing Society, pp. 69 – 72.

**5**. Mingxia Liu, Linsong Miao et al (2014), ’TwoStage Cost-Sensitive Learning forSoftware Defect Prediction’, IEEE Trans. Reliability, Vol. 63, pp.679-684.

**6**. P. Runeson and C. Wohlin (1998), ‘An Experimental Evaluation of an Experience-Based Capture-Recapture Method in Software Code Inspections’, Empirical Software Engineering., vol. 3, pp.381–406.

**7**. Q. Song, Z. Jia et al (2011), ‘A General Software Defect- Proneness Prediction Framework’, IEEE Transactions On Software Engineering, Vol. 37, pp.356-370.

**8**.Biffl, S. (2000). Using inspection data for defect estimation. IEEE Software, *17*(6), 36-43.

**9**.Petersson, H., &Wohlin, C. (1999, November). An empirical study of experience-based software defect content estimation methods. In Proceeding10th International Symposium onSoftware Reliability Engineering *(Cat. No. PR00443)* (pp. 126-135). IEEE.

**10**.Yalçıner, B., &Özdeş, M. (2019, September). 4th International Conference on ComputerScienceand Engineering IEEE

**11**. Victor R Basili, Lionel C. Briand, and Walce´lio L Melo. A validation of object-oriented design metrics as quality indicators. IEEE Transactions on software engineering, 22(10):751–761, 1996.

**12**. Victor R Basili, Lionel C. Briand, and Walce´lio L Melo. A validation of object-oriented design metrics as quality indicators. IEEE Transactions on software engineering, 22(10):751–761, 1996.

**13**.Evren Ceylan, F Onur Kutlubay, and Ayse B Bener. Software defect identification using machine learning techniques. In 32nd EUROMICRO Conference on Software Engineering and AdvancedApplications (EU- ROMICRO’06*)*, pages 240–247. IEEE, 2006.

**14**.Karim O Elish and Mahmoud O Elish. Predicting defect-prone software modules using support vector machines. Journal of Systems and Software, 81(5):649– 660, 2008.

**15**.Norman Fenton, Paul Krause, and Martin Neil. Software measurement: Uncertainty and causal modeling.

*IEEE* software, 19(4):116–122, 2002.

**16.**Lan Guo, Yan Ma, Bojan Cukic, and Harshinder Singh. Robust prediction of fault-proneness by random forests. In *15th* International Symposium on Software ReliabilityEngineering, pages 417–428. IEEE, 2004.

**17**.Taghi M Khoshgoftaar, Edward B Allen, and Jianyu Deng. Using regression trees to classify fault-prone software modules. *IEEE* Transactions on reliability, 51(4):455–462, 2002.

**18.**Taghi M Khoshgoftaar, Edward B Allen, John P Hude- pohl, and Stephen J Aud. Application of neural net- works to software quality modeling of a very large telecommunications system. *IEEE* Transactions on Neural Networks, 8(4):902–909, 1997.

**19.**Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *2011 33rd* International Conference on Software Engineering *(ICSE)*, pages 481–490. IEEE, 2011.

**20**.Yan Ma, Lan Guo, and Bojan Cukic. A statistical framework for the prediction of fault-proneness. In Advances in Machine Learning Applicationsin Software Engineering, pages 237–263. IGI Global, 2007.

**21.** Ruchika Malhotra. A systematic review of machine learning techniques for software fault prediction. Ap- plied Soft Computing, 27:504–518, 2015.

**22.** Rawat, M.S.; Dubey, S.K. Software defect prediction models for quality improvement: A literature study. *IJCSI Int. J. Comput. Sci. Issues* **2012**, *9*, 288–296. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Software+defect+prediction+models+for+quality+improvement:+A+literature+study&author=Rawat,+M.S.&author=Dubey,+S.K.&publication_year=2012&journal=IJCSI+Int.+J.+Comput.+Sci.+Issues&volume=9&pages=288%E2%80%93296)]

**23.** Aljarah, I.; Banitaan, S.; Abufardeh, S.; Jin, W.; Salem, S. Selecting discriminating terms for bug assignment: A formal analysis. In Proceedings of the 7th International Conference on Predictive Models in Software Engineering, Banff, AB, Canada, 20–21 September 2011; p. 12. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Selecting+discriminating+terms+for+bug+assignment:+A+formal+analysis&conference=Proceedings+of+the+7th+International+Conference+on+Predictive+Models+in+Software+Engineering&author=Aljarah,+I.&author=Banitaan,+S.&author=Abufardeh,+S.&author=Jin,+W.&author=Salem,+S.&publication_year=2011&pages=12)]

**24.** Fenton, N.E.; Neil, M. Software metrics: Roadmap. In Proceedings of the Conference on the Future of Software Engineering, Limerick, Ireland, 4–11 June 2000; pp. 357–370. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Software+metrics:+Roadmap&conference=Proceedings+of+the+Conference+on+the+Future+of+Software+Engineering&author=Fenton,+N.E.&author=Neil,+M.&publication_year=2000&pages=357%E2%80%93370)]

**25.** Fenton, N.; Bieman, J. *Software Metrics: A Rigorous and Practical Approach*; CRC Press: Boca Raton, FL, USA, 2014. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Software+Metrics:+A+Rigorous+and+Practical+Approach&author=Fenton,+N.&author=Bieman,+J.&publication_year=2014)]

**26**. Moser, R.; Pedrycz, W.; Succi, G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In Proceedings of the 2008 ACM/IEEE 30th International Conference on Software Engineering, Leipzig, Germany, 10–18 May 2008; pp. 181–190. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=A+comparative+analysis+of+the+efficiency+of+change+metrics+and+static+code+attributes+for+defect+prediction&conference=Proceedings+of+the+2008+ACM/IEEE+30th+International+Conference+on+Software+Engineering&author=Moser,+R.&author=Pedrycz,+W.&author=Succi,+G.&publication_year=2008&pages=181%E2%80%93190)]

**27.** Bhattacharya, P.; Iliofotou, M.; Neamtiu, I.; Faloutsos, M. Graph-based analysis and prediction for software evolution. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 419–429. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Graph-based+analysis+and+prediction+for+software+evolution&conference=Proceedings+of+the+2012+34th+International+Conference+on+Software+Engineering+(ICSE)&author=Bhattacharya,+P.&author=Iliofotou,+M.&author=Neamtiu,+I.&author=Faloutsos,+M.&publication_year=2012&pages=419%E2%80%93429)]

**28.** Abaei, G.; Selamat, A. A survey on software fault detection based on different prediction approaches. *Vietnam J. Comput. Sci.* **2014**, *1*, 79–95. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=A+survey+on+software+fault+detection+based+on+different+prediction+approaches&author=Abaei,+G.&author=Selamat,+A.&publication_year=2014&journal=Vietnam+J.+Comput.+Sci.&volume=1&pages=79%E2%80%9395&doi=10.1007/s40595-013-0008-z)] [[CrossRef](https://doi.org/10.1007/s40595-013-0008-z" \t "_blank)]

**29**. Wang, S.; Liu, T.; Tan, L. Automatically learning semantic features for defect prediction. In Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, 14–22 May 2016; pp. 297–308. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Automatically+learning+semantic+features+for+defect+prediction&conference=Proceedings+of+the+38th+International+Conference+on+Software+Engineering&author=Wang,+S.&author=Liu,+T.&author=Tan,+L.&publication_year=2016&pages=297%E2%80%93308)]

**30.**Clark, B.; Zubrow, D. How Good Is the Software*:* A Review of Defect Prediction Techniques; Sponsored by the US Department of Defense; InSoftware Engineering Symposium, Carreige Mellon University: Pittsburgh, PA, USA, 2001. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=How+Good+Is+the+Software:+A+Review+of+Defect+Prediction+Techniques&author=Clark,+B.&author=Zubrow,+D.&publication_year=2001)]

**31.**Hall, T.; Beecham, S.; Bowes, D.; Gray, D.; Counsell, S. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans. Softw. Eng.* **2012**, *38*, 1276–1304. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=A+systematic+literature+review+on+fault+prediction+performance+in+software+engineering&author=Hall,+T.&author=Beecham,+S.&author=Bowes,+D.&author=Gray,+D.&author=Counsell,+S.&publication_year=2012&journal=IEEE+Trans.+Softw.+Eng.&volume=38&pages=1276%E2%80%931304&doi=10.1109/TSE.2011.103)] [[CrossRef](https://doi.org/10.1109/TSE.2011.103" \t "_blank)]

**32.**Malhotra, R. A systematic review of machine learning techniques for software fault prediction. Appl. SoftComput*.* **2015**, *27*, 504–518. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=A+systematic+review+of+machine+learning+techniques+for+software+fault+prediction&author=Malhotra,+R.&publication_year=2015&journal=Appl.+Soft+Comput.&volume=27&pages=504%E2%80%93518&doi=10.1016/j.asoc.2014.11.023)] [[CrossRef](https://doi.org/10.1016/j.asoc.2014.11.023" \t "_blank)]

**33.**Menzies, T.; Milton, Z.; Turhan, B.; Cukic, B.; Jiang, Y.; Bener, A. Defect prediction from static code features: Current results, limitations, new approaches. *Autom. Softw. Eng.* **2010**, *17*, 375–407. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Defect+prediction+from+static+code+features:+Current+results,+limitations,+new+approaches&author=Menzies,+T.&author=Milton,+Z.&author=Turhan,+B.&author=Cukic,+B.&author=Jiang,+Y.&author=Bener,+A.&publication_year=2010&journal=Autom.+Softw.+Eng.&volume=17&pages=375%E2%80%93407&doi=10.1007/s10515-010-0069-5)] [[CrossRef](https://doi.org/10.1007/s10515-010-0069-5" \t "_blank)]

**34.**Li, Z.; Reformat, M. A practical method for the software fault-prediction. In Proceedings of the 2007 IEEE International Conference on Information Reuse and Integration, Las Vegas, IL, USA, 13–15 August 2007; pp. 659–666. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=A+practical+method+for+the+software+fault-prediction&conference=Proceedings+of+the+2007+IEEE+International+Conference+on+Information+Reuse+and+Integration&author=Li,+Z.&author=Reformat,+M.&publication_year=2007&pages=659%E2%80%93666)]

**35.**Vandecruys, O.; Martens, D.; Baesens, B.; Mues, C.; De Backer, M.; Haesen, R. Mining software repositories for comprehensible software fault prediction models. *J. Syst. Softw.* **2008**, *81*, 823–839. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Mining+software+repositories+for+comprehensible+software+fault+prediction+models&author=Vandecruys,+O.&author=Martens,+D.&author=Baesens,+B.&author=Mues,+C.&author=De+Backer,+M.&author=Haesen,+R.&publication_year=2008&journal=J.+Syst.+Softw.&volume=81&pages=823%E2%80%93839&doi=10.1016/j.jss.2007.07.034)] [[CrossRef](https://doi.org/10.1016/j.jss.2007.07.034" \t "_blank)]

**36**.Mendes-Moreira, J.; Soares, C.; Jorge, A.M.; Sousa, J.F.D. Ensemble approaches for regression: A survey. *ACM Comput. Surv. (CSUR)* **2012**, *45*, 10. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Ensemble+approaches+for+regression:+A+survey&author=Mendes-Moreira,+J.&author=Soares,+C.&author=Jorge,+A.M.&author=Sousa,+J.F.D.&publication_year=2012&journal=ACM+Comput.+Surv.+(CSUR)&volume=45&pages=10&doi=10.1145/2379776.2379786)] [[CrossRef](https://doi.org/10.1145/2379776.2379786" \t "_blank)]

**37.**Alsawalqah, H.; Faris, H.; Aljarah, I.; Alnemer, L.; Alhindawi, N. Hybrid SMOTE-Ensemble Approach for Software Defect Prediction. In Proceedings of the Computer Science On-Line Conference, Prague, Czech Republic, 26–29 April 2017; Springer: Cham, Switzerland, 2017; pp. 355–366. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Hybrid+SMOTE-Ensemble+Approach+for+Software+Defect+Prediction&conference=Proceedings+of+the+Computer+Science+On-Line+Conference&author=Alsawalqah,+H.&author=Faris,+H.&author=Aljarah,+I.&author=Alnemer,+L.&author=Alhindawi,+N.&publication_year=2017&pages=355%E2%80%93366)]

**38.**Rathore, S.S.; Kuamr, S. Comparative analysis of neural network and genetic programming for number of software faults prediction. In Proceedings of the 2015 National Conference on Recent Advances in Electronics & Computer Engineering (RAECE), Roorkee, India, 13–15 February 2015; pp. 328–332. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Comparative+analysis+of+neural+network+and+genetic+programming+for+number+of+software+faults+prediction&conference=Proceedings+of+the+2015+National+Conference+on+Recent+Advances+in+Electronics+&+Computer+Engineering+(RAECE)&author=Rathore,+S.S.&author=Kuamr,+S.&publication_year=2015&pages=328%E2%80%93332)]

**39.**Rathore, S.S.; Kumar, S. Predicting number of faults in software system using genetic programming. *Procedia Comput. Sci.* **2015**, *62*, 303–311. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Predicting+number+of+faults+in+software+system+using+genetic+programming&author=Rathore,+S.S.&author=Kumar,+S.&publication_year=2015&journal=Procedia+Comput.+Sci.&volume=62&pages=303%E2%80%93311&doi=10.1016/j.procs.2015.08.454)] [[CrossRef](https://doi.org/10.1016/j.procs.2015.08.454" \t "_blank)] [[Green Version](https://www.mdpi.com/2076-3417/10/5/1745)]

**40**.Rathore, S.S.; Kumar, S. An empirical study of some software fault prediction techniques for the number of faults prediction. *Soft Comput.* **2016**, *21*, 7417–7434. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=An+empirical+study+of+some+software+fault+prediction+techniques+for+the+number+of+faults+prediction&author=Rathore,+S.S.&author=Kumar,+S.&publication_year=2016&journal=Soft+Comput.&volume=21&pages=7417%E2%80%937434&doi=10.1007/s00500-016-2284-x)] [[CrossRef](https://doi.org/10.1007/s00500-016-2284-x" \t "_blank)]

**41.**Rathore, S.S.; Kumar, S. Linear and non-linear heterogeneous ensemble methods to predict the number of faults in software systems. *Knowl.-Based Syst.* **2017**, *119*, 232–256. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Linear+and+non-linear+heterogeneous+ensemble+methods+to+predict+the+number+of+faults+in+software+systems&author=Rathore,+S.S.&author=Kumar,+S.&publication_year=2017&journal=Knowl.-Based+Syst.&volume=119&pages=232%E2%80%93256&doi=10.1016/j.knosys.2016.12.017)] [[CrossRef](https://doi.org/10.1016/j.knosys.2016.12.017" \t "_blank)]

**42**.Shatnawi, R.; Li, W. The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *J. Syst. Softw.* **2008**, *81*, 1868–1882. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=The+effectiveness+of+software+metrics+in+identifying+error-prone+classes+in+post-release+software+evolution+process&author=Shatnawi,+R.&author=Li,+W.&publication_year=2008&journal=J.+Syst.+Softw.&volume=81&pages=1868%E2%80%931882&doi=10.1016/j.jss.2007.12.794)] [[CrossRef](https://doi.org/10.1016/j.jss.2007.12.794" \t "_blank)]

**43.**Sandhu, P.S.; Singh, S.; Budhija, N. Prediction of level of severity of faults in software systems using density based clustering. In Proceedings of the 2011 IEEE International Conference on Software and Computer Applications. IPCSIT, Kathmandu, Nepal, 1–2 July 2011. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Prediction+of+level+of+severity+of+faults+in+software+systems+using+density+based+clustering&conference=Proceedings+of+the+2011+IEEE+International+Conference+on+Software+and+Computer+Applications.+IPCSIT&author=Sandhu,+P.S.&author=Singh,+S.&author=Budhija,+N.&publication_year=2011)]

**44.**Menzies, T.; Greenwald, J.; Frank, A. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.* **2007**, *33*, 2–13. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Data+mining+static+code+attributes+to+learn+defect+predictors&author=Menzies,+T.&author=Greenwald,+J.&author=Frank,+A.&publication_year=2007&journal=IEEE+Trans.+Softw.+Eng.&volume=33&pages=2%E2%80%9313&doi=10.1109/TSE.2007.256941)] [[CrossRef](https://doi.org/10.1109/TSE.2007.256941" \t "_blank)]

**45**Suffian, M.D.M.; Ibrahim, S. A Prediction Model for System Testing Defects using Regression Analysis. *arXiv* **2014**, arXiv:1401.5830. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=A+Prediction+Model+for+System+Testing+Defects+using+Regression+Analysis&author=Suffian,+M.D.M.&author=Ibrahim,+S.&publication_year=2014&journal=arXiv)]

**46.**Koprinska, I.; Poon, J.; Clark, J.; Chan, J. Learning to classify e-mail. *Inf. Sci.* **2007**, *177*, 2167–2187. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Learning+to+classify+e-mail&author=Koprinska,+I.&author=Poon,+J.&author=Clark,+J.&author=Chan,+J.&publication_year=2007&journal=Inf.+Sci.&volume=177&pages=2167%E2%80%932187&doi=10.1016/j.ins.2006.12.005)] [[CrossRef](https://doi.org/10.1016/j.ins.2006.12.005" \t "_blank)]

**47.**Elish, K.O.; Elish, M.O. Predicting defect-prone software modules using support vector machines. *J. Syst. Softw.* **2008**, *81*, 649–660. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Predicting+defect-prone+software+modules+using+support+vector+machines&author=Elish,+K.O.&author=Elish,+M.O.&publication_year=2008&journal=J.+Syst.+Softw.&volume=81&pages=649%E2%80%93660&doi=10.1016/j.jss.2007.07.040)] [[CrossRef](https://doi.org/10.1016/j.jss.2007.07.040" \t "_blank)]

**48.**Huda, S.; Liu, K.; Abdelrazek, M.; Ibrahim, A.; Alyahya, S.; Al-Dossari, H.; Ahmad, S. An ensemble oversampling model for class imbalance problem in software defect prediction. *IEEE Access* **2018**, *6*, 24184–24195. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=An+ensemble+oversampling+model+for+class+imbalance+problem+in+software+defect+prediction&author=Huda,+S.&author=Liu,+K.&author=Abdelrazek,+M.&author=Ibrahim,+A.&author=Alyahya,+S.&author=Al-Dossari,+H.&author=Ahmad,+S.&publication_year=2018&journal=IEEE+Access&volume=6&pages=24184%E2%80%9324195&doi=10.1109/ACCESS.2018.2817572)] [[CrossRef](https://doi.org/10.1109/ACCESS.2018.2817572" \t "_blank)]

**49**Jiang, Y.; Cukic, B.; Ma, Y. Techniques for evaluating fault prediction models. *Empir. Softw. Eng.* **2008**, *13*, 561–595. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Techniques+for+evaluating+fault+prediction+models&author=Jiang,+Y.&author=Cukic,+B.&author=Ma,+Y.&publication_year=2008&journal=Empir.+Softw.+Eng.&volume=13&pages=561%E2%80%93595&doi=10.1007/s10664-008-9079-3)] [[CrossRef](https://doi.org/10.1007/s10664-008-9079-3" \t "_blank)]

**50.**Sun, Z.; Song, Q.; Zhu, X. Using coding-based ensemble learning to improve software defect prediction. *IEEE Trans. Syst. Man Cybern. Part C (Appl. Rev.)* **2012**, *42*, 1806–1817. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Using+coding-based+ensemble+learning+to+improve+software+defect+prediction&author=Sun,+Z.&author=Song,+Q.&author=Zhu,+X.&publication_year=2012&journal=IEEE+Trans.+Syst.+Man+Cybern.+Part+C+(Appl.+Rev.)&volume=42&pages=1806%E2%80%931817&doi=10.1109/TSMCC.2012.2226152)] [[CrossRef](https://doi.org/10.1109/TSMCC.2012.2226152" \t "_blank)]

**51**.El Emam, K.; Benlarbi, S.; Goel, N.; Rai, S.N. Comparing case-based reasoning classifiers for predicting high risk software components. *J. Syst. Softw.* **2001**, *55*, 301–320. [[Google Scholar](https://scholar.google.com/scholar_lookup?title=Comparing+case-based+reasoning+classifiers+for+predicting+high+risk+software+components&author=El+Emam,+K.&author=Benlarbi,+S.&author=Goel,+N.&author=Rai,+S.N.&publication_year=2001&journal=J.+Syst.+Softw.&volume=55&pages=301%E2%80%93320&doi=10.1016/S0164-1212(00)00079-0)] [[CrossRef](https://doi.org/10.1016/S0164-1212(00)00079-0" \t "_blank)]

Lourdes Pelayo and Scott Dick (2007), ‘Applying Novel Resembling Strategies to Software Defect Prediction,’ in proc.North Amr.Fuzzy Inf. Processing Society, pp. 69 – 72.

**52**. Mingxia Liu, Linsong Miao et al (2014), ’TwoStage Cost-Sensitive Learning forSoftware Defect Prediction’, IEEE Trans. Reliability, Vol. 63, pp.679-684.

**53** P. Runeson and C. Wohlin (1998), ‘An Experimental Evaluation of an Experience-Based Capture-Recapture Method in Software Code Inspections’, Empirical Software Engineering., vol. 3, pp.381–406.

**54**. Q. Song, Z. Jia et al (2011), ‘A General Software Defect- Proneness Prediction Framework’, IEEE Transactions On Software Engineering, Vol. 37, pp.356-370.

**55**. J.Zheng (2010), ’Cost-sensitive boosting neural networks for software defect prediction’, Expert Systems with Appl., vol.37, pp. 4537-4543