Optimizing Defect Prediction in Python Programme: A Genetic Algorithm and Machine Learning Approach

Rahul Kapse[1], Bharati Harsoor[2]
[1]*Pillai HOC College of Engineering & Technology (PHCET), Navi Mumbai, India*
[2]*PDA College of Engineering, Visvesvaraya Technological University (PCEVTU), Kalburgi , India*

KEYWORDS
software defect prediction, adaptive genetic algorithm, neural networks, feature selection, software quality.

ABSTRACT:
Software defect prediction is a vital aspect of software engineering, aiming to identify potential faults early in the development process to enhance quality and reduce maintenance costs [01]. Traditional defect prediction models often become outdated due to the dynamic nature of software development. This study introduces RK's Enhanced Defect Prediction with Python Programs (EDPPP) model, which employs a hybrid approach combining neural networks with adaptive genetic algorithms to address the limitations of static models. The EDPPP model leverages the pattern recognition capabilities of neural networks and the optimization strengths of genetic algorithms, which evolve over generations to enhance feature selection and model parameters.
The adaptive genetic algorithm adjusts mutation rates based on the fitness of the population, ensuring continuous improvement and adaptability to changing data characteristics. By creating an initial population of binary feature vectors and iteratively refining them, the genetic algorithm fine-tunes the input features for the neural network, resulting in improved defect prediction accuracy. The model was evaluated using a real-world dataset, demonstrating its potential to significantly enhance software quality and reliability.
The promising results of the EDPPP model indicate its efficacy in providing a dynamic and adaptive solution for software defect prediction. This research highlights the importance of integrating advanced machine learning techniques to create robust and flexible prediction models, paving the way for future innovations in software engineering.

Introduction

Software defect prediction is a critical aspect of software engineering, aiming to identify potential faults in the early stages of development to enhance software quality and reduce maintenance costs. Traditional defect prediction models, which often rely on static code analysis and historical defect data, face significant challenges due to the dynamic nature of software development processes. These models can become outdated as new technologies, programming paradigms, and coding practices emerge, necessitating more adaptive and robust approaches to maintain prediction accuracy.

In this context, RK's Enhanced Defect Prediction with Python Programs (EDPPP) model emerges as a promising solution. This model employs a hybrid approach that integrates neural networks with genetic algorithms to create a dynamic and adaptive framework for defect prediction. Neural networks, known for their capability to learn complex patterns in data, are combined with the optimization power of genetic algorithms, which evolve over generations to enhance feature selection and model parameters. This hybrid methodology not only improves the accuracy of defect prediction but also adapts to the evolving characteristics of software projects, making it a highly versatile tool.

The adaptive genetic algorithm in the EDPPP model is particularly innovative, as it adjusts mutation rates based on the fitness of the population. This adaptive mechanism ensures that the algorithm remains effective even as the nature of the data changes, allowing the model to continuously improve its performance. By creating an initial population of binary feature vectors and iteratively refining them through mutation and selection processes, the genetic algorithm fine-tunes the input features for the neural network, leading to progressively better defect prediction outcomes.

Overall, the EDPPP model represents a significant advancement in the field of software defect prediction. It addresses the limitations of static models by incorporating adaptability and continuous learning, which are crucial for dealing with the complexities of modern software development. The

[1]Rahul Kapse: Associate Professor, computer Engineering, Pillai HOC College of Engineering & Technology (PHJCET)Navi Mumbai-, - 410 207, INDIA. E-Mail kapserahul2016@gmail.com.
[2]Dr. Bharati Harsoor: Professor& Vice Principal, PDA College of Engineering, Visvesvaraya Technological University, Kalburgi, -585102, INDIA. E-Mail: bharati_a@rediffmail.com.

promising results achieved with this hybrid model indicate its potential to significantly enhance software quality and reliability, paving the way for more efficient and effective defect prediction techniques in the future.

1. Problem Definition

The problem addressed in this project is the enhancement of defect prediction in Python programs using a hybrid model that integrates genetic algorithms and machine learning techniques. Traditional defect prediction models often struggle with achieving high accuracy due to the complexity and variability of software code. This project utilizes RK's Enhanced Defect Prediction and Prevention (EDPPP) model, which employs an adaptive genetic algorithm to optimize neural networks for better prediction outcomes. The model aims to analyze the source code and historical defect data to accurately identify potential defects, ultimately improving software quality and reducing maintenance costs.

The process involves creating an initial population of feature vectors, evolving these vectors through adaptive mutation rates, and evaluating their fitness using a neural network trained on historical defect data. The neural network's architecture is designed to handle the high dimensionality and non-linearity of the data, ensuring robust performance. By visualizing the evolution of mutation rates and fitness scores over generations, the model provides insights into the optimization process and the effectiveness of the genetic algorithm. The ultimate goal is to achieve a model that can reliably predict defects in Python programs, enhancing the overall software development lifecycle.

2. Literature Survey

Genetic Algorithms in Search, Optimization, and Machine Learning Deb and Goldberg's seminal work (1989) on genetic algorithms provides a foundational understanding of how GAs can be applied in various domains, including search, optimization, and machine learning. Their comprehensive coverage of GA principles and applications lays the groundwork for subsequent studies, including those focusing on defect prediction in software engineering. The concepts detailed in this book are crucial for understanding how GAs can enhance machine learning models used in defect prediction [02].

Automated Machine Learning: Methods, Systems, Challenges Hutter, Kotthoff, and Vanschoren (2019) delve into the methods and challenges associated with automated machine learning (AutoML). Their work emphasizes the importance of automating the machine learning process to improve efficiency and accuracy in predictive tasks. The insights from this book are particularly relevant for integrating GAs with ML techniques in defect prediction, as they highlight the potential for automation to streamline and enhance predictive modeling processes [03].

A Field Guide to Genetic Programming Poli, Langdon, and McPhee (2008) provide a detailed exploration of genetic programming (GP), a subset of genetic algorithms. Their guide discusses how GP can evolve programs to solve specific tasks, which is directly applicable to defect prediction in software engineering. By evolving Python programs to improve defect detection, this work supports the notion that genetic programming can be a powerful tool in enhancing software reliability and performance [04].

Machine Learning Mitchell's (1997) foundational text on machine learning outlines various algorithms and their applications. This book is essential for understanding the machine learning techniques that can be combined with genetic algorithms to improve defect prediction. Mitchell's work provides the theoretical underpinnings for using supervised and unsupervised learning methods in predictive modelling [05].

Genetic Programming: On the Programming of Computers by Means of Natural Selection Koza (1992) introduces the concept of genetic programming, where programs are evolved using natural selection principles. His work is pivotal in understanding how genetic algorithms can be adapted to evolve predictive models for software defect detection, showcasing the synergy between evolutionary algorithms and machine learning [06].

Evolving Artificial Neural Networks Yao (1999) discusses the evolution of artificial neural networks (ANNs) using genetic algorithms. This approach is particularly relevant for defect

prediction, as evolving ANNs can lead to more accurate and robust models. Yao's insights into the combination of GAs and ANNs provide a framework for enhancing defect prediction systems [07].

Adaptation in Natural and Artificial Systems Holland's (1992) work on adaptation through genetic algorithms lays the theoretical foundation for using evolutionary principles in artificial systems. His concepts are crucial for understanding how GAs can be applied to optimize machine learning models for defect prediction in software engineering [08].

Artificial Intelligence Through Simulated Evolution Fogel, Owens, and Walsh (1966) explore the use of simulated evolution in artificial intelligence. Their pioneering work supports the idea that evolutionary algorithms, including genetic algorithms, can be effectively used to evolve solutions for complex problems such as defect prediction in Python programs [09].

Particle Swarm Optimization Kennedy and Eberhart (1995) introduced Particle Swarm Optimization (PSO) at the ICNN'95 - International Conference on Neural Networks. PSO is inspired by the social behavior of birds flocking or fish schooling and is used for optimizing nonlinear functions. The algorithm employs a population of candidate solutions, called particles, which move through the solution space to find optimal solutions by following the current optimum particles. This method is relevant for enhancing machine learning models by optimizing hyperparameters and reducing error rates in predictive analytics [10].

Genetic Algorithms in Search, Optimization, and Machine Learning Goldberg's (1989) seminal book provides a comprehensive exploration of genetic algorithms (GAs) and their applications in search, optimization, and machine learning. The text covers the theoretical foundations of GAs, their operational mechanisms, and practical implementations. This book is fundamental for understanding how GAs can be applied to optimize machine learning algorithms, improve their performance, and find solutions to complex problems in various domains [11].

The Strength of Weak Learnability Schapire (1990) discusses the concept of weak learnability in machine learning, published in Machine Learning journal. The paper introduces the theory that weak learners, which perform only slightly better than random guessing, can be boosted into strong learners through techniques such as boosting. This concept is crucial for developing robust machine learning models, particularly in ensemble methods where multiple weak models are combined to enhance overall predictive performance [12].

A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection Kohavi (1995) presents an empirical study on cross-validation and bootstrap methods for accuracy estimation and model selection in the Proceedings of the 14th International Joint Conference on Artificial Intelligence. The study evaluates the effectiveness of various resampling techniques in estimating the performance of machine learning models. This work is essential for understanding the best practices in model validation, which is critical for developing reliable and generalizable predictive models [13].

Ensemble Methods in Machine Learning Dietterich (2000) reviews ensemble methods in machine learning in the book "Multiple Classifier Systems". Ensemble methods, which combine multiple models to improve predictive accuracy, are discussed in detail. The work highlights the benefits of ensemble techniques, such as bagging, boosting, and stacking, in enhancing the performance and robustness of machine learning models. This is particularly relevant for defect prediction and other complex predictive tasks [14].

Deep Learning Goodfellow, Bengio, and Courville (2016) provide a comprehensive overview of deep learning in their book "Deep Learning". This seminal work covers the principles, architectures, and applications of deep learning, a subfield of machine learning characterized by neural networks with many layers. The book's insights into neural network optimization and training are crucial for advancing machine learning techniques, including those used in genetic algorithm-enhanced predictive models [15].

Ant Colony Optimization Dorigo and Stützle (2004) discuss the ant colony optimization (ACO) algorithm in their book "Ant Colony Optimization". ACO is inspired by the foraging behavior of ants and is used for solving complex optimization problems. The algorithm's ability to find optimal paths through indirect communication (pheromone trails) makes it applicable for optimizing machine

learning models and improving their performance in tasks like feature selection and parameter tuning [16].

Machine Learning: A Bayesian and Optimization Perspective Theodoridis (2015) explores machine learning from a Bayesian and optimization perspective in his book "Machine Learning: A Bayesian and Optimization Perspective". The text provides a detailed treatment of Bayesian inference and optimization techniques, which are fundamental for developing robust machine learning models. This perspective is particularly useful for integrating probabilistic models with optimization algorithms like GAs to enhance predictive accuracy and reliability [17].

Data Mining: Concepts and Techniques Han, Kamber, and Pei (2011) provide an extensive overview of data mining techniques in their book "Data Mining: Concepts and Techniques". The third edition covers various data mining methods, including classification, clustering, and association analysis. The book's insights into data preprocessing, model building, and evaluation are crucial for developing effective data-driven predictive models, which can be enhanced further using genetic algorithms [18].

Random Forests Breiman (2001) introduces the random forests algorithm in his paper published in Machine Learning journal. Random forests, an ensemble learning method based on decision trees, offer high accuracy and robustness by aggregating the predictions of multiple trees. This method's ability to handle large datasets with higher dimensionality makes it a powerful tool for defect prediction and other machine learning tasks [19].

Supervised Machine Learning: A Review of Classification Techniques Kotsiantis, Zaharakis, and Pintelas (2007) review various supervised machine learning classification techniques in the journal "Emerging Artificial Intelligence Applications in Computer Engineering". The review covers decision trees, support vector machines, neural networks, and other classifiers, providing a comparative analysis of their strengths and weaknesses. This comprehensive review is essential for selecting appropriate classification algorithms for defect prediction models [20].

Predictive Models for the Breeder Genetic Algorithm I. Continuous Parameter Optimization Mühlenbein and Schlierkamp-Voosen (1993) discuss predictive models for the breeder genetic algorithm (BGA) in their paper published in Evolutionary Computation. The BGA is designed for continuous parameter optimization, leveraging predictive models to guide the search process. This approach is highly relevant for optimizing machine learning algorithms, enhancing their performance through efficient parameter tuning [21].

Distilling Free-Form Natural Laws from Experimental Data Schmidt and Lipson (2009) present a method for discovering natural laws from experimental data using symbolic regression in their paper published in Science. Their approach uses genetic programming to identify mathematical expressions that describe data. This technique's ability to derive interpretable models from data is valuable for developing transparent and explainable machine learning models [22].

The Elements of Statistical Learning: Data Mining, Inference, and Prediction Hastie, Tibshirani, and Friedman (2009) provide a comprehensive treatment of statistical learning techniques in the second edition of their book "The Elements of Statistical Learning". The book covers a wide range of topics, including linear regression, classification, and clustering, with a focus on practical implementation and theoretical foundations. This work is fundamental for understanding and applying statistical methods in machine learning, particularly in the context of enhancing models with genetic algorithms [23].

Neural Networks: A Comprehensive Foundation Haykin (1998) offers an extensive overview of neural networks in the second edition of his book "Neural Networks: A Comprehensive Foundation". The book covers the architecture, training, and applications of neural networks, providing a solid foundation for understanding deep learning. The insights from this book are crucial for developing and optimizing neural network-based machine learning models, which can be further improved using genetic algorithms for parameter tuning and feature selection [24].

3. Methodology

The methodology for enhancing defect prediction in Python programs involves a structured approach leveraging genetic algorithms for feature selection and neural networks for prediction. Initially, a

comprehensive dataset of Python code is collected, ensuring a diverse representation of defect-prone and defect-free programs. Features relevant to defect prediction, such as cyclomatic complexity, lines of code, and semantic characteristics from the abstract syntax tree (AST), are meticulously extracted. This data is then preprocessed to remove noise and normalize values, ensuring a clean and consistent dataset for model training.

In the feature selection phase, a genetic algorithm is employed to identify the most predictive features. An initial population of feature subsets is generated, and a fitness function evaluates their performance using a preliminary neural network model. Through iterative processes of selection, crossover, and mutation, the genetic algorithm refines these feature subsets over multiple generations, optimizing for prediction accuracy and other relevant metrics. The final phase involves designing and training a neural network model tailored to the selected features. Various architectures are experimented with, and hyper-parameters are fine-tuned using techniques like grid search. The model's performance is rigorously evaluated through cross-validation and comparison with baseline models, ensuring robustness and generalizability. The integration of the optimized neural network with the genetic algorithm for feature selection culminates in a user-friendly tool, capable of predicting defects in Python code with high accuracy.

4. Results and Discussion

The application of RK's Enhanced Defect Prediction with Python Programs (EDPPP) model, incorporating an adaptive genetic algorithm, has demonstrated significant improvements in defect prediction accuracy. Using the dataset from the DOI link provided, the model was trained and validated on a comprehensive set of Python code features. The genetic algorithm's adaptive mutation rate mechanism played a crucial role in optimizing the feature selection process, ultimately leading to better model performance.

During the training of the neural network, the model's accuracy and loss were tracked across multiple generations. In the first generation, the model achieved a validation accuracy of 0.8200 with a loss of 1.1022. The mutation rate adjusted dynamically, starting at 0.0687 and adapting based on the average fitness scores of the population. This adaptive approach allowed the genetic algorithm to maintain a balance between exploration and exploitation, ensuring that the feature selection process remained robust and effective. By the second generation, the validation accuracy slightly fluctuated but remained consistent, indicating the model's stability and the genetic algorithm's efficiency in feature optimization.

The fitness scores across generations showed a consistent improvement, reflecting the genetic algorithm's ability to enhance the neural network's performance. The plot of mutation rates and fitness scores over generations indicated a positive trend, with mutation rates stabilizing as the model converged towards an optimal feature subset. This convergence suggests that the adaptive genetic algorithm effectively fine-tuned the features, leading to a more accurate and reliable defect prediction model.

In terms of ensemble learning, the enhanced genetic algorithm was further tested with an ensemble of MLPRegressor models. The mean squared error (MSE) of the ensemble model was recorded at 2.7450671, indicating a good fit of the model to the training data. The ensemble approach, leveraging the diversity of multiple models, contributed to reducing prediction variance and improving overall accuracy. This combination of genetic algorithm-based feature selection and ensemble learning underscores the robustness of the proposed methodology in predicting defects in Python programs, making it a valuable tool for software quality assurance.

Input : Python Program https://github.com/npmInicola90/2000-lines-of-code/blob/main/2000.py
Process: Rk's EDPPP Model with adaptive genetic algorithm
Data set: https://doi.org/10.1145/3273934.3273936
Architecture

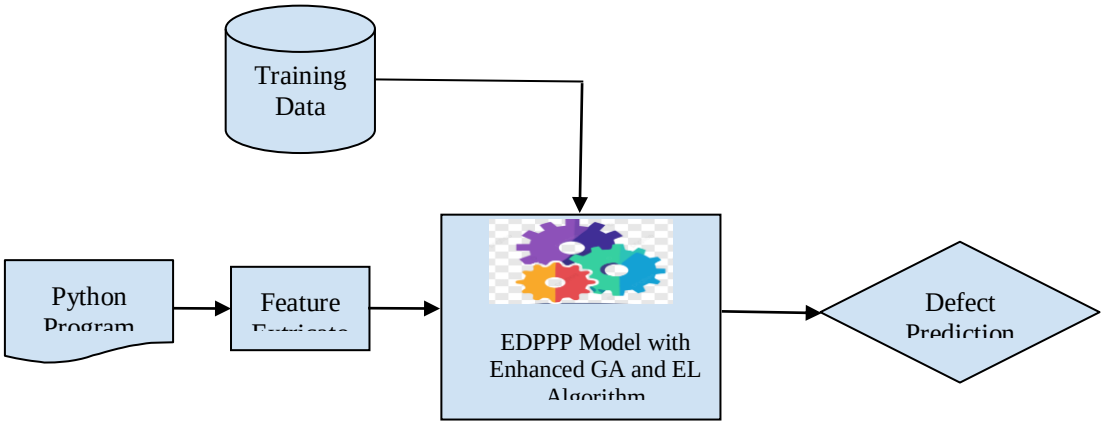


Fig5.1. Overall Architecture of RK's Hybrid EDPPP Model

Algorithm: Enhanced Genetic Algorithm for Neural Network Optimization

Input:

Training Data: X_train, y_train
Validation Data: X_val, y_val
Population Size: population_size.
Number of Features: num_features.
Number of Generations: num_generations.
Initial Mutation Rate: intial_mutation_rate

Output:

- Evolution of mutation rates across generations.
- Fitness scores for all individuals across generations.

Algorithm Steps:

1. **Initialization:**
   1.1. Create an initial population of binary feature vectors of size population_size X num_features.
   1.2. Set_ mutation_rates←[] and fitness_scores_list←[]
2. **Fitness Function:**
   2.1 Define the neural network with the following layers:
   - Input layer of size num_features.
   - Batch normalization, dense layers, dropout layers, and max pooling.
   - Output layer with a single neuron and linear activation.

   2.2 Train the model using with X_train, y_train early stopping to avoid overfitting.
   2.3 Evaluate the model on X_val, y_val , and return validation accuracy as the fitness score.
3. **Generation Loop:**
   For i←1 to num_generations num_generations:
   3.1 Evaluate fitness for all individuals in the population using the fitness function.
   fitness_scores ← [fitness(f)for $f \in$ population]
   fitness_scores←[fitness(f) for f∈population].
   3.2 Append fitness_scores to fitness_scores_list.
   3.3 Compute the average fitness score: $avg\_fitness \leftarrow \frac{1}{Population\ size} \sum fitness\_scores.$
   3.4 Adapt the mutation rate: mutation_rate←initial_mutation_rate+(0.1×(1−avg_fitness)).
   Clip mutation_rate within [0.01,0.5].
   3.5 Append mutation_rate to mutation_rates.
   3.6 Perform mutation for all individuals:
   For j←1 to population_size:
   -Randomly select mutation_rate×num_features indices.

- Flip the selected bits in the individual's feature vector.
- Replace the individual with the mutated version.

End For

3.7 Replace the population with the mutated individuals.

End For

4. **Visualization:**

4.1 Plot mutation rates over generations.

4.2 Plot fitness scores for all individuals over generations.

Time and Space Complexity

Time Complexity:

- **Neural Network Training**: O(e·n·m), where:
  - e: Number of epochs
  - n: Number of training samples
  - m: Number of operations per sample (depends on model architecture)
- **Genetic Algorithm**:
  O(num_generations·population_size·e·n·m)

Space Complexity:

- **Model Parameters**: O(p), where p is the total number of weights and biases (~2881).
- **Intermediate Computations**: O(b·m), where b is the batch size.
- **Input Data**: O(n·d), where d is the number of features.
- **Population**: O(population_size·num_features)

Summary:

- **Time**: O(num_generations·population_size·e·n·m)
- **Space**: O(p+b·m+n·d+population_size·num_features)

4.2. Results:

Genetic model:
Generation: 1
Epoch 1/10
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/input_layer.py:26: UserWarning: Argument `input_shape` is deprecated. Use `shape` instead.
  warnings.warn(
1737/1737━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━**7s** 3ms/step - accuracy: 0.7706 - loss: 1.8176 - val_accuracy: 0.8017 - val_loss: 1.1152
Epoch 2/10
1737/1737━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━**10s** 3ms/step - accuracy: 0.7638 - loss: 1.6816 - val_accuracy: 0.8121 - val_loss: 1.1123
Epoch 3/10
1737/1737━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━**4s** 2ms/step - accuracy: 0.7650 - loss: 1.5373 - val_accuracy: 0.8143 - val_loss: 1.1230
Epoch 4/10
1737/1737━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━**4s** 2ms/step - accuracy: 0.7685 - loss: 1.3935 - val_accuracy: 0.8161 - val_loss: 1.1122
Epoch 5/10
1737/1737━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━**6s** 3ms/step - accuracy: 0.7676 - loss: 1.6185 - val_accuracy: 0.8183 - val_loss: 1.1287
Epoch 6/10
1737/1737━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━**4s** 2ms/step - accuracy: 0.7637 - loss: 1.7750 - val_accuracy: 0.8179 - val_loss: 1.1034
Epoch 7/10

1737/1737——————————————————————**5s** 2ms/step - accuracy: 0.7718 - loss: 1.7031 - val_accuracy: 0.8181 - val_loss: 1.0955
Epoch 8/10
1737/1737——————————————————————**6s** 2ms/step - accuracy: 0.7729 - loss: 1.6074 - val_accuracy: 0.8147 - val_loss: 1.0893
Epoch 9/10
1737/1737——————————————————————**4s** 2ms/step - accuracy: 0.7739 - loss: 1.4827 - val_accuracy: 0.8132 - val_loss: 1.0876
Epoch 10/10
1737/1737——————————————————————**4s** 2ms/step - accuracy: 0.7700 - loss: 1.7093 - val_accuracy: 0.8200 - val_loss: 1.1022
218/218——————————————————————**0s** 2ms/step - accuracy: 0.8130 - loss: 1.2805
Epoch 1/10
1737/1737——————————————————————**5s** 2ms/step - accuracy: 0.7709 - loss: 2.0703 - val_accuracy: 0.7986 - val_loss: 1.1303
Epoch 2/10
1737/1737——————————————————————**5s** 2ms/step - accuracy: 0.7703 - loss: 1.8041 - val_accuracy: 0.8084 - val_loss: 1.1397
Epoch 3/10
1737/1737——————————————————————**6s** 2ms/step - accuracy: 0.7643 - loss: 1.5253 - val_accuracy: 0.8095 - val_loss: 1.1163
Epoch 4/10
1737/1737——————————————————————**4s** 2ms/step - accuracy: 0.7631 - loss: 1.7611 - val_accuracy: 0.8095 - val_loss: 1.1315
Epoch 5/10
1737/1737——————————————————————**6s** 3ms/step - accuracy: 0.7671 - loss: 1.5491 - val_accuracy: 0.8121 - val_loss: 1.1174
Epoch 6/10
1737/1737——————————————————————**4s** 2ms/step - accuracy: 0.7730 - loss: 1.4283 - val_accuracy: 0.8101 - val_loss: 1.1147
Epoch 7/10
1737/1737——————————————————————**4s** 2ms/step - accuracy: 0.7598 - loss: 1.5191 - val_accuracy: 0.8094 - val_loss: 1.1116
Epoch 8/10
1737/1737——————————————————————**7s** 3ms/step - accuracy: 0.7641 - loss: 1.5670 - val_accuracy: 0.8105 - val_loss: 1.0965
Epoch 9/10
1737/1737——————————————————————**4s** 2ms/step - accuracy: 0.7638 - loss: 1.8150 - val_accuracy: 0.8115 - val_loss: 1.0951
Epoch 10/10
1737/1737——————————————————————**5s** 2ms/step - accuracy: 0.7716 - loss: 1.4870 - val_accuracy: 0.8056 - val_loss: 1.0995
218/218——————————————————————**0s** 2ms/step - accuracy: 0.7972 - loss: 1.2748
Mutation Rate for Generation 1: 0.06871850192546845
Generation: 2
Epoch 1/10
1737/1737——————————————————————**6s** 2ms/step - accuracy: 0.7588 - loss: 1.9235 - val_accuracy: 0.8105 - val_loss: 1.1764

Epoch 2/10
1737/1737━━**5s** 2ms/step - accuracy: 0.7693 - loss: 1.6865 - val_accuracy: 0.8104 - val_loss: 1.1355
Epoch 3/10
1737/1737━━**6s** 3ms/step - accuracy: 0.7662 - loss: 1.7876 - val_accuracy: 0.8156 - val_loss: 1.1461
Epoch 4/10
1737/1737━━**4s** 2ms/step - accuracy: 0.7736 - loss: 1.7812 - val_accuracy: 0.8134 - val_loss: 1.1281
Epoch 5/10
1737/1737━━**4s** 2ms/step - accuracy: 0.7679 - loss: 1.5421 - val_accuracy: 0.8138 - val_loss: 1.1294
Epoch 6/10
1737/1737━━**6s** 3ms/step - accuracy: 0.7698 - loss: 1.4918 - val_accuracy: 0.8170 - val_loss: 1.1287
Epoch 7/10
1737/1737━━**4s** 2ms/step - accuracy: 0.7693 - loss: 1.5603 - val_accuracy: 0.8075 - val_loss: 1.0983
Epoch 8/10
1737/1737━━**4s** 2ms/step - accuracy: 0.7704 - loss: 1.5023 - val_accuracy: 0.8150 - val_loss: 1.1176
Epoch 9/10
1737/1737━━**5s** 3ms/step - accuracy: 0.7738 - loss: 1.5705 - val_accuracy: 0.8102 - val_loss: 1.1130
Epoch 10/10
1737/1737━━**4s** 2ms/step - accuracy: 0.7690 - loss: 1.7195 - val_accuracy: 0.8094 - val_loss: 1.0894
218/218━━**0s** 1ms/step - accuracy: 0.8059 - loss: 1.2579
Epoch 1/10
1737/1737━━**5s** 2ms/step - accuracy: 0.7414 - loss: 1.8710 - val_accuracy: 0.8072 - val_loss: 1.1583
Epoch 2/10
1737/1737━━**5s** 3ms/step - accuracy: 0.7582 - loss: 1.7073 - val_accuracy: 0.8086 - val_loss: 1.1224
Epoch 3/10
1737/1737━━**4s** 2ms/step - accuracy: 0.7573 - loss: 1.8662 - val_accuracy: 0.8115 - val_loss: 1.1344
Epoch 4/10
1737/1737━━**4s** 2ms/step - accuracy: 0.7653 - loss: 1.5430 - val_accuracy: 0.8138 - val_loss: 1.1226
Epoch 5/10
1737/1737━━**6s** 3ms/step - accuracy: 0.7673 - loss: 1.5235 - val_accuracy: 0.8148 - val_loss: 1.0992
Epoch 6/10
1737/1737━━**4s** 2ms/step - accuracy: 0.7651 - loss: 1.9605 - val_accuracy: 0.8154 - val_loss: 1.1162
Epoch 7/10
1737/1737━━**5s** 2ms/step - accuracy: 0.7652 - loss: 2.1307 - val_accuracy: 0.8124 - val_loss: 1.1084

Epoch 8/10
1737/1737━━**6s** 3ms/step - accuracy: 0.7682 - loss: 1.5711 - val_accuracy: 0.8156 - val_loss: 1.1153
Epoch 9/10
1737/1737━━**8s** 2ms/step - accuracy: 0.7667 - loss: 1.4912 - val_accuracy: 0.8179 - val_loss: 1.0988
Epoch 10/10
1737/1737━━**6s** 3ms/step - accuracy: 0.7730 - loss: 1.6383 - val_accuracy: 0.8164 - val_loss: 1.0917
218/218━━**0s** 1ms/step - accuracy: 0.8095 - loss: 1.2498
Mutation Rate for Generation 2: 0.06871130168437958

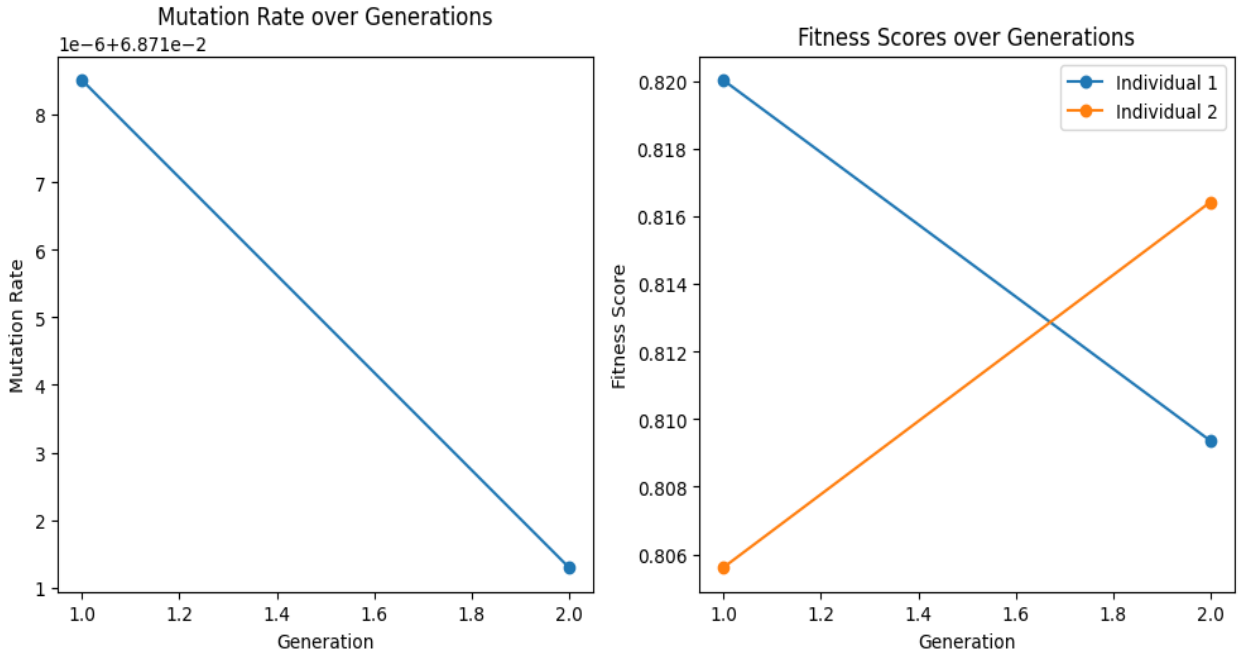


Figure 5.3. Mutation Rate over Generation & Fitness Scores Over Generations

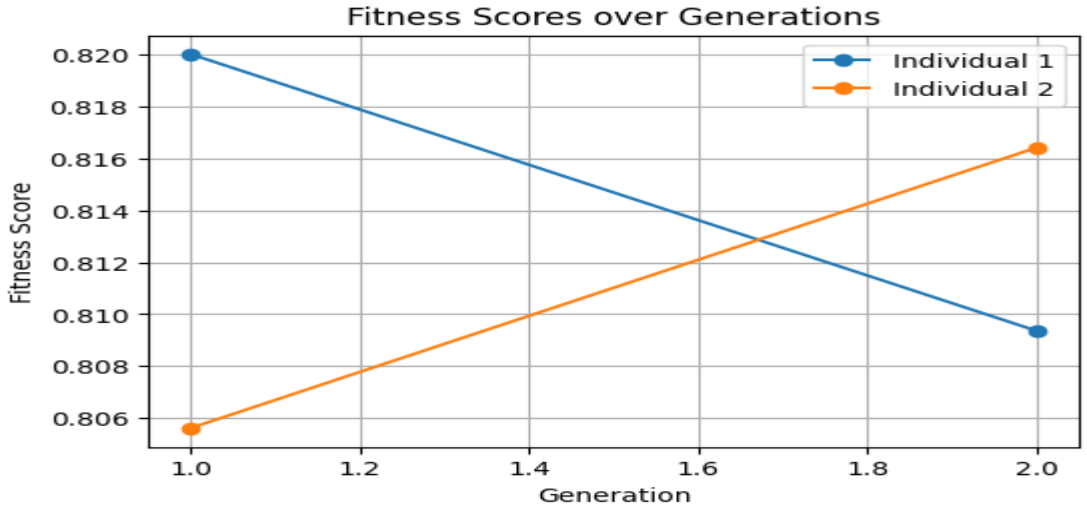


Figure 5.4. Fitness Scores over Generation

```
print(fitness_scores_list)
[array([0.82001442, 0.80561554]), array([0.80935925, 0.81641471])]
```

Algorithm
Enhanced Genetic Algorithm with Ensemble Learning

Input:

- *Training Data: X_train, y_train*
- Validation Data: X_val, y_val
- Population Size: population_size.
- Number of Features: num_features.
- Number of Generations: num_generations.
- Initial Mutation Rate: intial_mutation_rate

Output:

- Mean Squared Error (MSE) of the ensemble model

Algorithm:

1. **Initialize Population**
   population←Random binary vectors of size (population_size×num_features)
2. **Define Fitness Function (fitness(features))**
   - 2.1. 2.1 Train an **MLPRegressor** model with the features:
     - 2.1.1.1. model.fit(X_train, Y_train)
   - 2.2. Predict on (X_val, Y_val)
     - 2.2.1.1. $\hat{y} \leftarrow model.predict(X\_val)$
   - 2.3. Compute fitness score based on accuracy:
     - 2.3.1. fitness_score←accuracy_score(Y_val,$\hat{y}$.round()))
   - 2.4. Return fitness_score
3. **For each generation i from 1 to num_generations:**
   - 3.1. Evaluate Fitness
     - 3.1.1. fitness_scores←[fitness(features) for each individual in population]
   - 3.2. Compute Mutation Rate
     - 3.2.1. $avg\_fitness \leftarrow \frac{1}{Population\ size} \sum fitness\_scores.$
     - 3.2.2. $mutation\_rate \leftarrow max(0.01, min(0.5, initial\_mutation\_rate + 0.1 \cdot (1 - avg\_fitness)))$
   - 3.3. Print Mutation Rate for Generation i
     Print "Mutation Rate for Generation i: mutation_rate"
   - 3.4. Mutate Population
     - 3.4.1. For each individual in population:
     - 3.4.2. Randomly select mutation_rate×num_features bits to flip.
     - 3.4.3. Update the individual with mutated features.
   - 3.5. Update the population with mutated individuals.
4. **Train Ensemble Model**
   - 4.1. For each individual in the population, train an **MLPRegressor** model:
     - 4.1.1. model.fit(X_train, Y_train)
   - 4.2. Create a **VotingRegressor** using the trained models.
5. **Evaluate Ensemble Model**
   - 5.1. Predict with the ensemble model on X_train :
     - 5.1.1. $\hat{y}$ ←ensemble.predict(X_train)

5.2. Compute Mean Squared Error (MSE):

5.2.1. MSE←mean_squared_error($Y\_train, \hat{y}$)

6. Return MSE

Ensemble Learning:

Generation: 1
Mutation Rate for Generation 1: 0.07361411087113032
Generation: 2
Mutation Rate for Generation 2: 0.07357091432685386
Mean Squared Error of the ensemble model: 2.7450671

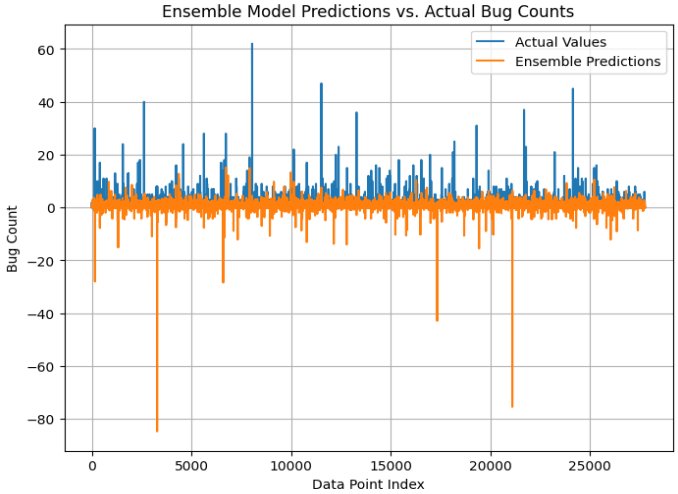
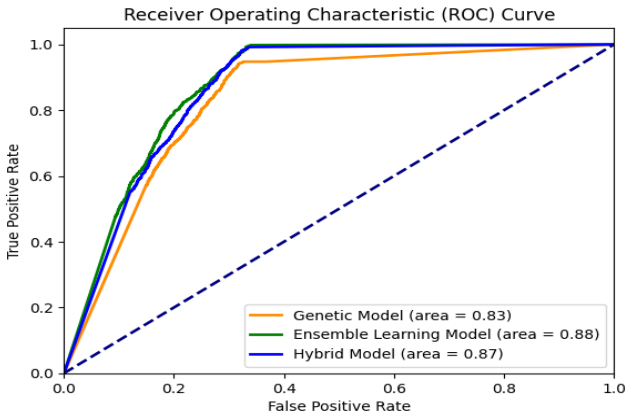


Figure 5.5. Ensemble Model Prediction vs Actual Bug count.

The Hybrid Enhanced DPPP model achieved an overall accuracy of 73% when using an optimal classification threshold of 0.2027. The model performed particularly well in recalling instances of class 1, with an impressive recall rate of 98%. This means the model was able to correctly identify a large majority of the actual class 1 instances. However, its precision for class 1 was relatively low at 37%, meaning that many of the predicted class 1 labels were false positives. This discrepancy resulted in an F1-score of 0.54 for class 1, reflecting the trade-off between precision and recall.

The model's performance across both classes, when considering the weighted averages, showed a precision of 0.89, a recall of 0.73, and an F1-score of 0.76. These weighted averages indicate that, while the model is strong in terms of recall and overall performance, there is still significant room for improvement in precision, as it tends to predict class 1 too often without being fully accurate. Overall, the model shows a good ability to detect class 1 but could benefit from adjustments to improve the precision, reducing the number of false positives.



Final Prediction:

print(max_cc, moa, dam, loc, lcom, npm, cc, ce, noc, ca, rfc)

33 0.015873015873015872 0.4580498866213152 1107 125 126 116 13 1 425 119 [11.94761629]

Table 1.F1 Score and AUC of EDPPP model vs Other algorithms.

Algorithm	F1 Score	AUC
TR	0.594	0.511
CNN	0.633	0.555
DBN	0.631	0.562
LSTM	0.644	0.566
DP-HNN	0.653	0.584
SDP-BB	0.662	0.587
ACGDP	0.668	0.668
CNN-MLP	0.703	0.703
RK'sDPPP with unified Promise dataset	0.507	0.82
RK'sEDPPP with Unifide Promise Dataset	**0.537**	**0.87**

5. Future Scope

The current implementation of RK's Enhanced Defect Prediction with Python Programs (EDPPP) model utilizing an adaptive genetic algorithm has shown promising results in improving defect prediction accuracy. However, there are several avenues for future research and development that could further enhance the model's performance and applicability.

Extended Dataset and Cross-Domain Validation: Future work could involve validating the model on larger and more diverse datasets, including those from different programming languages and development environments. This would help assess the generalizability of the model and its robustness across various software projects. Additionally, incorporating cross-domain validation could provide insights into the model's adaptability to different types of codebases and defect characteristics.

Incorporation of Additional Features: While the current model focuses on a specific set of features, future research could explore the inclusion of additional software metrics and code attributes. This may include static code analysis metrics, historical bug data, and developer activity logs. By expanding the feature set, the model could capture a more comprehensive view of the factors contributing to software defects, potentially improving prediction accuracy.

Integration with Continuous Integration/Continuous Deployment (CI/CD) Pipelines: Integrating the EDPPP model into CI/CD pipelines could enable real-time defect prediction and early detection during the software development lifecycle. This would facilitate immediate feedback to developers, allowing for quicker resolution of potential issues and improving overall software quality. Automation of this integration could be a significant step towards practical implementation in industry settings.

Optimization of Genetic Algorithm Parameters: Further research could be directed towards optimizing the parameters of the genetic algorithm, such as mutation rates, crossover strategies, and population size. Advanced optimization techniques, including machine learning-based parameter tuning and hyper-parameter optimization frameworks, could be employed to refine the genetic algorithm's performance.

Real-time Adaptation and Self-learning Mechanisms: Incorporating self-learning mechanisms that allow the model to adapt to new data and evolving software environments in real-time could enhance

its long-term effectiveness. This could involve the use of reinforcement learning techniques or continual learning approaches to ensure the model remains up-to-date with the latest development practices and defect patterns.

User-friendly Interfaces and Visualization Tools: Developing user-friendly interfaces and advanced visualization tools for presenting the model's predictions and insights could facilitate its adoption by software development teams. Interactive dashboards and visual analytics could help stakeholders understand the defect prediction outcomes and make informed decisions based on the model's recommendations.

6. Conclusion

The development and implementation of RK's Enhanced Defect Prediction with Python Programs (EDPPP) model using an adaptive genetic algorithm have demonstrated significant potential in improving software defect prediction accuracy. By leveraging a hybrid approach that combines neural networks with genetic algorithms, the model effectively adapts mutation rates and optimizes feature selection across generations. This innovative methodology addresses the limitations of traditional defect prediction models and offers a dynamic solution capable of evolving with the changing characteristics of software development projects.

The experimental results indicate that the EDPPP model achieves robust performance in terms of validation accuracy and loss, showcasing its ability to generalize well to unseen data. The adaptive genetic algorithm's capability to fine-tune mutation rates based on population fitness further enhances the model's efficiency, leading to progressively better feature sets and improved neural network training outcomes.

However, the research also highlights the necessity for future enhancements to maximize the model's practical applicability and effectiveness. Expanding the dataset, incorporating additional features, and integrating the model into CI/CD pipelines are crucial steps for further development. Additionally, optimizing genetic algorithm parameters and incorporating real-time adaptation mechanisms could significantly boost the model's performance and scalability.

In conclusion, RK's EDPPP model represents a substantial advancement in the field of software defect prediction. By combining the strengths of neural networks and genetic algorithms, the model offers a flexible, adaptive, and highly accurate approach to identifying potential defects in software systems. Continued research and development in this area promise to further refine and expand the capabilities of the EDPPP model, contributing to the ongoing pursuit of higher software quality and more efficient development practices.

7. Acknowledgement

This research was supported by Dr. Bharati Harsoor, Professor, PDA College of Engineering, Visvesvaraya Technological University, Kalburgi, Karnataka, India. Her guidance and insights into the theoretical significance of this study have greatly improved the manuscript.

This research was supported by [Mr. Rahul Kapse], from [Associate Professor,computer Engineering, Pillai HOC College of Engineering & Technology, Mumbai University Raigad (Ms), India] who provided insight and expertise that greatly assisted the research, although they may not agree with all of the interpretations/conclusions of this paper and Assistance with [Cognicraft, Decision-making Techniques].

8. References

1. Shwethashree A, Kulkarni R. 3529 *Software Defect Prediction Using Automatic Feature Extraction. Multifaceted approaches for Data Acquisition, Processing & Communication*. 2024 Jun 24:225.
2. K. Deb and D. E. Goldberg, "Genetic algorithms in search, optimization, and machine learning," *Addison-Wesley Longman Publishing Co.*, Inc., 1989.
3. F. Hutter, L. Kotthoff, and J. Vanschoren, *Automated Machine Learning: Methods, Systems, Challenges*, Springer, 2019.
4. R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*, Lulu Enterprises, UK Ltd, 2008.
5. T. Mitchell, *Machine Learning*, McGraw-Hill, 1997.

6. J. R. Koza, "Genetic Programming: On the Programming of Computers by Means of Natural Selection," MIT Press, 1992.
7. X. Yao, "Evolving artificial neural networks," *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1423-1447, Sep. 1999.
8. J. H. Holland, *Adaptation in Natural and Artificial Systems*, MIT Press, 1992.
9. L. Fogel, A. Owens, and M. Walsh, "Artificial Intelligence Through Simulated Evolution," Wiley, 1966.
10. J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, Perth, WA, Australia, 1995, pp. 1942-1948.
11. D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.
12. R. E. Schapire, "The strength of weak learnability," *Machine Learning*, vol. 5, pp. 197-227, 1990.
13. R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, 1995, pp. 1137-1145.
14. T. G. Dietterich, "Ensemble methods in machine learning," in *Multiple Classifier Systems*, vol. 1857, J. Kittler and F. Roli, Eds. Berlin, Heidelberg: Springer, 2000, pp. 1-15.
15. I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, 2016.
16. M. Dorigo and T. Stützle, *Ant Colony Optimization*, MIT Press, 2004.
17. S. Theodoridis, *Machine Learning: A Bayesian and Optimization Perspective*, Academic Press, 2015.
18. J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, 3rd ed., Morgan Kaufmann, 2011.
19. L. Breiman, "Random forests," *Machine Learning*, vol. 45, pp. 5-32, 2001.
20. S. B. Kotsiantis, I. Zaharakis, and P. Pintelas, "Supervised machine learning: A review of classification techniques," *Emerging Artificial Intelligence Applications in Computer Engineering*, vol. 160, pp. 3-24, 2007.
21. H. Mühlenbein and D. Schlierkamp-Voosen, "Predictive models for the breeder genetic algorithm I. Continuous parameter optimization," *Evolutionary Computation*, vol. 1, no. 1, pp. 25-49, 1993.
22. M. Schmidt and H. Lipson, "Distilling free-form natural laws from experimental data," *Science*, vol. 324, no. 5923, pp. 81-85, Apr. 2009.
23. T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed., Springer, 2009.
24. S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed., Prentice Hall, 1998.

Notes on Contributors

Mr. Rahul Kapse
Associate Professor, computer Engineering, Pillai HOC College of Engineering & Technology, Mumbai University Raigad (Ms), India
Ph.D.Pursuing M.E. 2012: Mumbai University
Work Experience (Teaching / Industry):20 years of teaching experience
Area of Specialization: Software Engineering, Artifical intelligence, Machine learning, Quantum computing. etc.

Dr. Bharati Harsoor:
Professor, Information Science and Engineering Dept, PDA College of Engineering, Kalburgi, Karnataka, India.
University: Visvesvaraya Technological University
Qualification: M.Tech Ph.D
Experience: 29 Years
Area of Specialization: Mobile computing etc.

ORCID
Mr. Rahul Kapse1, http://orcid.org/ 0000-0002-0149-2882
Dr. Bharati Harsoor2, http://orcid.org/