Journal of Education for Pure Science
Vol.15, No. 1 (2025)
DOI: https://doi.org/ 10.32792/jeps.v15i1.658
Website: jceps.utq.edu.iq Email: jceps@eps.utq.edu.iq

Prioritization of Modules to Reduce Software Testing Time and Costs Using Evolutionary Algorithms and the Kilo Lines of Code(KLOC)Method

Tahseen Sabbar Rhaef Al-Saadawi
Thi Qar Governorate Police Command
Corresponding email: Tahseniraqi86@gmail.com

Received 19 / 1 /2025,Accepted 28/2 /2025, Published 01/03/2025

Abstract:

Software testing is a critical and resource-intensive phase in the Software Development Life Cycle (SDLC), often consuming up to 50% of development costs. This study introduces an innovative module prioritization approach that optimizes testing time and cost while maintaining software quality. By integrating the KLOC (Kilo Lines of Code) metric with advanced genetic algorithms (GA) and Deep General Regression Neural Networks (DGRNN), the proposed method efficiently allocates testing resources to prioritize critical modules. This ensures comprehensive testing within constrained budgets, significantly reducing both time and cost. The methodology was implemented using MATLAB simulations, where DGRNN was compared against traditional machine learning models, including Support Vector Regression (SVR), k-Nearest Neighbors (k-NN), and Random Forest (RF). The results demonstrate that the GA-KLOC approach reduces testing time by up to 30% and lowers costs by 25% while maintaining high defect detection accuracy of over 90%. This research highlights the effectiveness of evolutionary algorithms and deep learning in optimizing software testing processes, offering a practical solution to enhance software quality, reduce errors, and improve cost efficiency in software development. The study also addresses the gap in module-level defect density prediction by leveraging DGRNN, providing a robust framework for future research in software testing optimization.

Keywords: Module prioritization, time reduction, software testing cost reduction, evolutionary algorithms, KLOC method

1-Introduction

Software testing is a critical and resource-intensive component of the Software Development Life Cycle (SDLC), accounting for up to 50% of development costs [1]. Cost reductions can be achieved through test automation, case prioritization, and suite minimization [2]. Advances in artificial intelligence (AI) provide opportunities to optimize testing processes, especially through machine learning and deep learning, which analyze test cases to enhance efficiency [3]. However, manual testing remains prevalent in safety-critical systems due to reliance on human judgment [4]. Dependencies among integrated test cases affect test execution, often leading to sequential failures when improperly prioritized [5]. Classifying test cases as dependent or independent remains challenging, particularly given imbalanced data distributions [6]. Accurate and efficient solutions are required to optimize testing and reduce costs.

This study introduces a module prioritization method using genetic algorithms and the KLOC method to allocate testing time and resources. Critical components are prioritized, ensuring comprehensive testing within constrained budgets. The model divides testing time proportionally based on priority, balancing efficiency and thoroughness. This approach addresses the limited research on cost-efficient testing in the region, offering a practical solution to enhance

software quality and development processes. The objective of this paper is to propose a method for reducing cost and time in software testing using a genetic algorithm. The process is divided into three phases.

Research on defect density prediction is divided into two main directions. The first direction focuses on analyzing the relationship between defects and various software metrics. Rahmani and Khazanchi [7] examined the relationship between defect density and three metrics (number of downloads, number of developers, and software size) using data from 44 software projects. They found that two of the four proposed hypotheses were statistically significant. Mandhan et al. [8] demonstrated that seven design and code metrics could significantly predict defect density. Nagappan and Ball [9] used code churn metrics to predict defect density in the early stages and showed that these metrics were sufficient for prediction through statistical correlations and machine learning models. Verma and Kumar [10] proposed six hypotheses to study the relationship between defect density and various metrics, of which four were statistically validated.

The second direction focuses on developing defect density prediction models using existing defect datasets. Kotlubay et al. [11] applied Decision Tree and Naïve Bayes methods to predict defect density in NASA projects, finding that Decision Tree models performed better than regression models. López Martín et al. [12] introduced a novel method called Transformation of K-Nearest Neighbor Distance Minimization (TKDM), which outperformed other machine learning models, including Support Vector Regression (SVR). Huang et al. [13] proposed the DeepJIT model, a deep learning approach that identifies defects by extracting embedded features from commit messages and code changes. Zhao et al. [14] developed DeepSim, a technique for measuring functional similarities in code, leveraging neural network classifiers for defect prediction. Hu et al. [14] employed convolutional neural networks (CNNs) to extract semantic features from source code and bug reports for defect localization.

Importantly, no previous study has utilized module-level defect density; prior research focused exclusively on project-level defect density. Furthermore, none of the studies used deep learning for defect density prediction. Existing deep learning approaches primarily targeted defect prediction by classifying software modules as defective or non-defective. These gaps serve as the primary motivation for this research, which introduces an advanced Deep GRNN model for module-level defect density prediction.

This study makes several key contributions:

1. It introduces a novel genetic algorithm-based module prioritization method for software testing, optimizing resource allocation and reducing costs.

2. It proposes a deep learning-based approach for module-level defect density prediction, filling an existing research gap.

3. It enhances defect prediction accuracy by leveraging an advanced Deep GRNN model, improving software quality and reliability.

4. By addressing these critical areas, this research provides a comprehensive and scientifically grounded approach to optimizing software testing and defect prediction processes.

2- Methodology

2.1 Module Prioritization Using Genetic Algorithm:

Traditional research methods in software testing can be costly and time-intensive. To address this, a genetic algorithm was employed to optimize cost and time efficiency. The General Regression Neural Network (GRNN) was enhanced into a Deep GRNN (DGRNN) by adding three dense hidden layers before the pattern layer to extract deeper features. The DGRNN consists of seven layers: an input layer, five hidden layers (including dropout for overfitting prevention), and an output layer. Using Radial Basis Function (RBF) activation, the network identifies intricate patterns and generates outputs like defect density, with weight matrices updated via gradient descent.

2.2 Component-Based Prioritization Technique for Time and Cost Optimization:

The prioritization technique employing the genetic algorithm follows several steps, as illustrated in Figure 1. This

method ensures efficient optimization of both time and cost, making the software testing process more accurate and resource-efficient. It focuses on prioritizing modules within large-scale software projects based on qualitative criteria, which are often vaguely defined for decision-makers. Fortunately, various benchmark datasets are available through software repositories, allowing us to evaluate and generalize our findings. However, only datasets containing defect count variables are considered since defect density cannot be calculated without this information. We gathered 28 publicly available datasets from three primary sources: the AEEEM repository, SOFTLAB, and MORPH.

Our study also examines the impact of data distribution on defect density prediction models. Relevant attributes, such as KLOC (thousands of lines of code), customer priority, and the average KLOC of all modules, were considered for module prioritization.

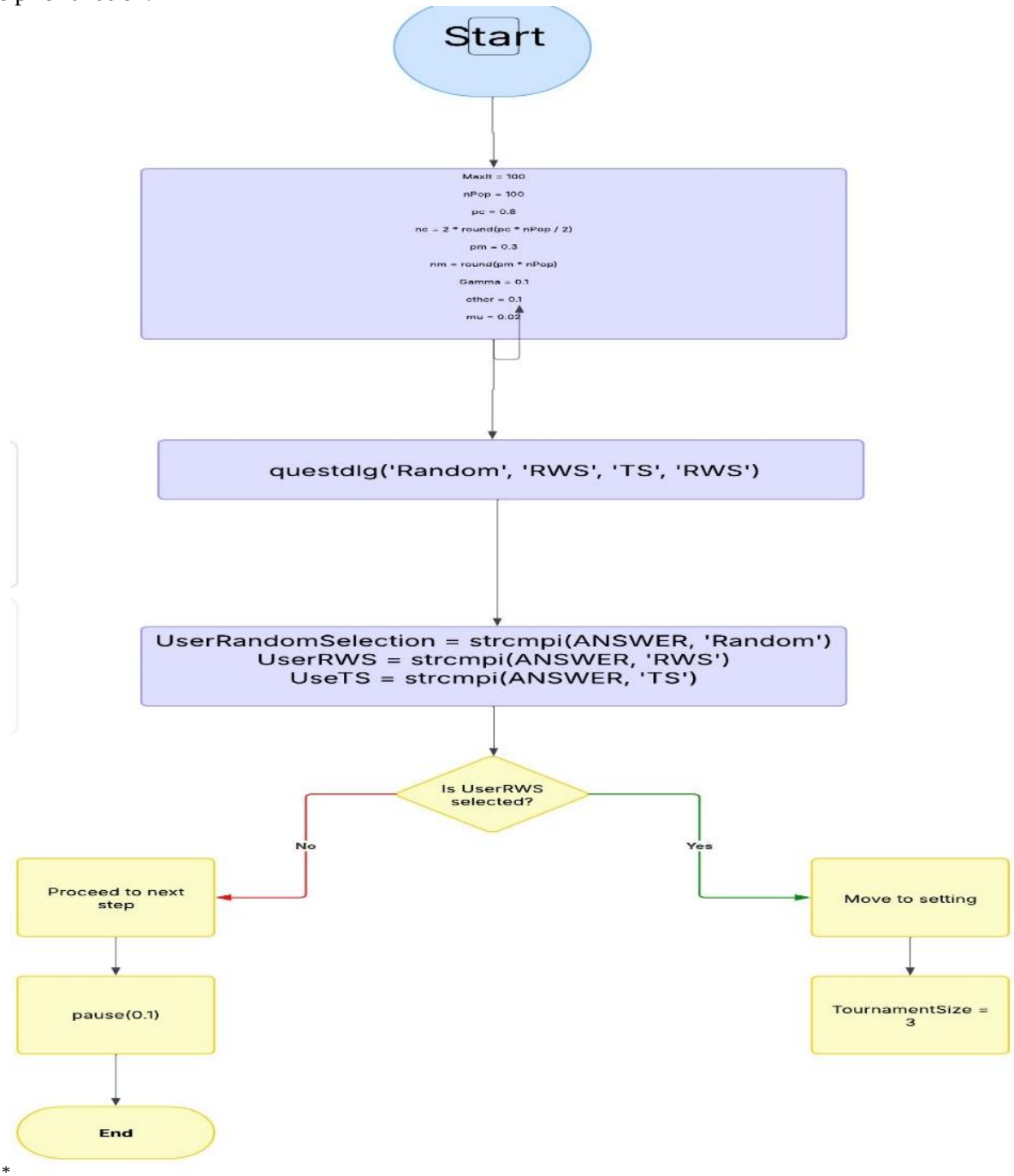
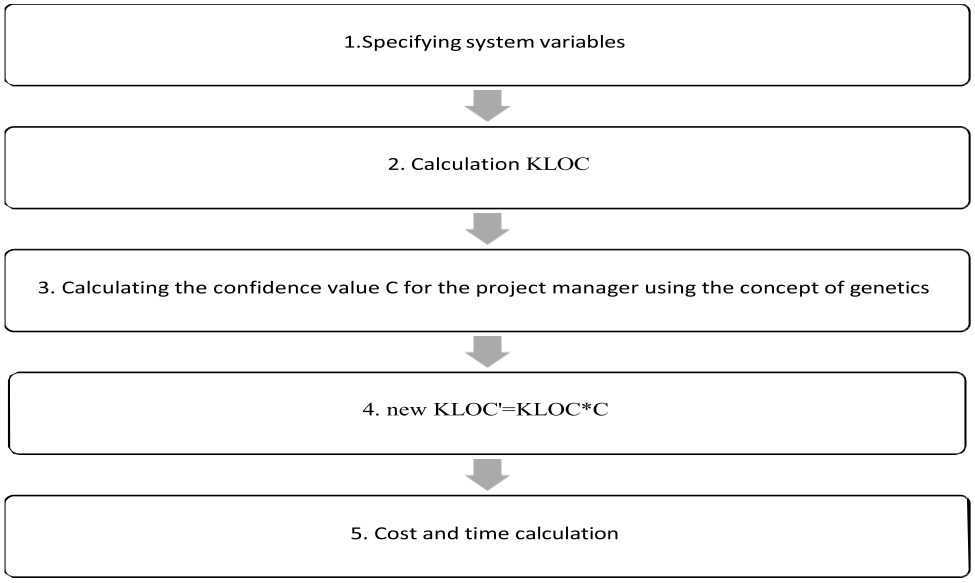


Figure 1:Component-Based Prioritization Technique for Time and Cost Optimization

2.3 Software Testing

It is widely recognized that the performance of a software system is closely related to the time spent on testing. More time dedicated to testing typically leads to better system performance and lower defect-fixing costs, as issues are less expensive to address during testing than in operation. However, excessive testing can delay the product's release, increasing overall costs. The challenge lies in finding the optimal balance, identifying the release time that minimizes overall costs by preventing both insufficient testing and unnecessary delays. This concept is illustrated in Figure 2.

Figure 2. Software testing steps.



Here is the algorithm for the described steps:

Step 1: Define System Variables

1. Define system variables:
   - Time variable (Time)
   - Cost variable (Cost)
   - Constant parameters (e.g., fixed costs, fixed time constraints)
   - Optional parameters (e.g., user-defined constraints)
2. Use genetic algorithms to optimize these parameters for minimizing cost and time while accommodating the number of users entering the system.

Step 2: Estimate KLOC

1. Calculate the KLOC (thousands of lines of code) for the project/unit.
2. Determine the organization type (e.g., small, medium, large).
3. Estimate the time and cost based on the total number of users and KLOC.

Step 3: Calculate Project Manager Trust (C)

1. Evaluate the project manager's trustworthiness (C) based on their experience and familiarity with the project.
2. Use genetic algorithms to combine maximum precision in calculating C.

Step 4: Calculate Adjusted KLOC

1. Compute the adjusted KLOC (KLOC'):

KLOC′=KLOC×CKLOC′=KLOC×C
where C is the project manager's trust factor.

Step 5: Cost and Time Calculation

1. Initialize the genetic algorithm parameters:
   - Population size (nPop)
   - Maximum iterations (MaxIt)
   - Crossover rate (nc)
   - Mutation rate (nm)
   - Tournament size (TournamentSize)
   - Mutation parameters (mu, etha)
   - Variable bounds (VarMin, VarMax)
2. For each iteration (it = 1:MaxIt):
   - **Selection**:
     - If using Roulette Wheel Selection (UserRWS):
       - Calculate selection probabilities:

P=exp⁡(−β×cost/worst cost)∑(exp⁡(−β×cost/worst cost))P=∑(exp(−β×cost/worst cost))exp(−β×cost/worst cost)

       - Select parents using Roulette Wheel Selection.
     - If using Tournament Selection (UseTS):
       - Select parents using Tournament Selection.
     - If using Random Selection (UseRandomSelection):
       - Select parents randomly.
   - **Crossover**:
     - Perform arithmetic crossover to generate offspring:

Offspring=γ×Parent1+(1−γ)×Parent2Offspring=γ×Parent1+(1−γ)×Parent2

     - Evaluate the cost of the offspring using the cost function.
   - **Mutation**:
     - Apply mutation to a subset of the population:

Mutated Position=Position+etha×randnMutated Position=Position+etha×randn

     - Evaluate the cost of the mutated individuals.
   - **Population Integration**:
     - Combine the current population, offspring, and mutated individuals.
     - Sort the population based on cost.
     - Truncate the population to maintain the original size (nPop).
   - **Update Best Solution**:
     - Track the best solution (BestSol) and its cost (BestCost).
     - Update the worst cost (WorstCost).
   - **Display Iteration Information**:
     - Print the iteration number, number of function evaluations (NFE), and best cost.
   - **Visualize Solutions**:
     - Plot the best solution at each iteration.

Finding Optimal Cost and Time

We analyzed the relationship between defect density and the number of samples in the datasets, finding that smaller datasets often have a lower defect density. This confirms that larger modules generally exhibit a higher defect density. Additionally, we explored the effect of data dispersion on defect density prediction, dividing datasets into four groups based on the interquartile range of SR.

Limitations and Improvements:

Despite its innovativeness, this methodology presents complexities and potential risks such as overfitting, slow convergence, and limited generalizability. Moreover, scalability and the optimization of the genetic algorithm-based prioritization technique remain areas that need addressing for broader applications. Furthermore, the image-based algorithm representations should be rewritten as text-based algorithms, as the provided software is not in the form of proper algorithms and must be converted into appropriate algorithmic representations.

Algorithm

Below is an algorithm based on the translated text:

Algorithm: Analyze Defect Density and SR Using Genetic Algorithm

Input:

Datasets with defect density and SR values.

Genetic algorithm parameters (population size, crossover rate, mutation rate, etc.).

Output:

Optimized list of users with associated cost and time.

Analysis of defect density and SR relationships.

Steps:

1. **Initialize Parameters**:
   Set population size, crossover rate, mutation rate, and maximum iterations.
   Define the fitness function to evaluate cost and time.
2. **Load Datasets**:
   Load 28 datasets with defect density and SR values.
   Divide datasets into four groups based on the interquartile range of SR.
3. **Plot Scatter Plot**:
   Plot a scatter plot between defect density and the number of samples.
   Analyze the relationship between module size and defect density.
4. **Plot Histogram**:
   Plot a histogram of SR values for all datasets.
   Identify the median SR value and classify datasets into high, medium, and low SR groups.
5. **Genetic Algorithm**:
   Initialize a population of solutions (user lists with associated cost and time).
   For each iteration:

   a. **Selection**:
      Use roulette wheel selection or tournament selection to select parents.

   b. **Crossover**:

Perform arithmetic crossover to generate offspring.

c. **Mutation**:
Apply mutation to introduce diversity in the population.

d. **Evaluation**:
Evaluate the fitness of each solution using the fitness function.

e. **Update Population**:
Sort the population based on fitness.
Retain the best solutions and discard the worst.

f. **Check Termination**:
Stop if the maximum number of iterations is reached or the solution converges.

6. **Extract Optimized User List**:
   Extract the best solution (user list) with the lowest cost and time.
7. **Analyze Results**:
   Compare defect density and SR values across datasets.
   Identify trends and relationships between module size, defect density, and SR.
8. **Output Results**:
   Display the optimized user list, cost, and time.
   Provide insights into defect density and SR relationships.

limitations by using a genetic algorithm to extract detailed user, cost, and time data, enabling precise analysis.

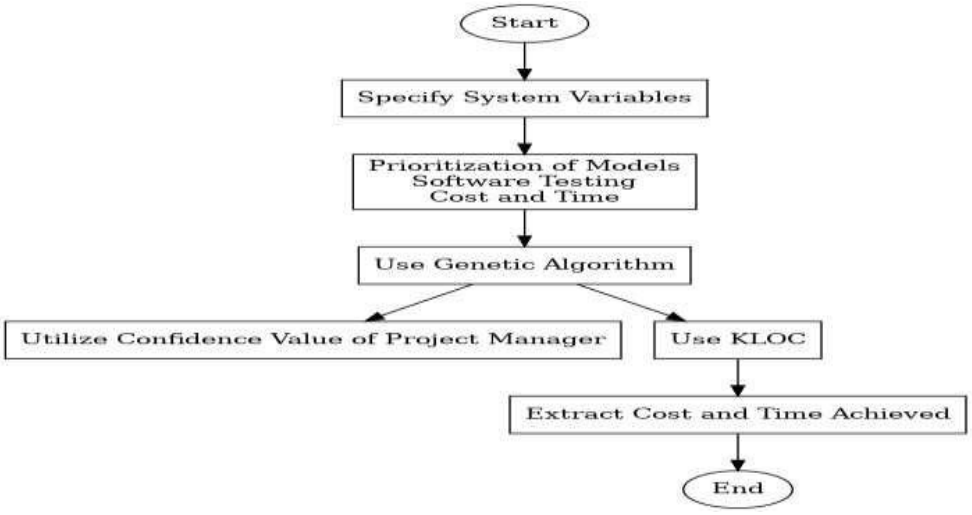


Figure 3: Proposed flowchart.

3- Results

3.1 Simulation results

The simulation results provide insights into the efficiency and effectiveness of the proposed approach. By running multiple simulation scenarios, we analyzed key performance metrics, including time, cost, and accuracy,

to evaluate the impact of different parameters on software testing. The results were obtained using MATLAB, where we implemented a genetic algorithm to optimize the allocation of resources and users in the testing process.

Simulation Environment:

In this study, we used the MATLAB simulator, which allows us to leverage the best possible scenarios using the rules and algorithms available in the software. Since MATLAB is a comprehensive simulator for conducting research in both engineering and non-engineering fields, we were able to determine the optimal time and cost based on the number of users in software testing using the genetic algorithm.

Simulation Data Collection

Software testing topics are primarily theoretical, with data collected through document research and reviewing articles from sources like ACM, IEEE, and Springer. Preprocessing involved transforming numerical variables using min-max scaling, applying one-hot encoding for categorical variables, and handling outliers via box plots. Observations with extreme values were removed, and missing data were imputed using mean or mode imputation, ensuring reliability and reducing biases in the predictive model.

In this study, we focused on optimizing software systems by leveraging cost and time functions, with the help of a genetic algorithm. To evaluate the effectiveness of our proposed DGRNN model, we compared it against several well-known machine learning regression techniques. These included Support Vector Regression, k-Nearest Neighbors, Multi-Layer Perceptron, Extreme Gradient Boosting (XGB), AdaBoost, Random Forest, Multiple Linear Regression, and GRNN. The goal was to see how our model stacks up against these established methods in terms of performance and accuracy. For all machine learning algorithms, default configuration parameters were used. The results allowed us to accurately identify the optimal conditions for the system.

Table 1. Simulation variables.

System Variables	Performance
Amin	Minimum demand or output value
Amax	Maximum output value
Bmin	Minimum input capacity of the system
Bmax	Maximum input capacity of the system
Bmean	Average input power
Cmin	Minimum cost
Cmax	Maximum cost
A	Final value derived from average parameters
B	Average input capacity of the system
C	Average cost
I	Input
J	Output

Table 2. Simulation functions

Function	Performance in the Program
NFE	Used for calculating software testing time. In this function, time is considered constant to reduce software costs. The higher the time, the cost increases incrementally.
Model Name	The name of the model to be implemented and applied.
Best Cost	Used for calculating the cost in software testing.

Capv	Resolves any errors in the system that occur during operations. Since we use a genetic algorithm, this function is included as a helper function to prevent and address errors when they arise.

Sudoku program to get cost and time:

```
global NFE;
NFE=0;
model=SelectModel();
CostFunction=@(xhat) MyCost(xhat,model);
VarSize=[model.I model.J];
arMin=0;
VarMax=1;
```

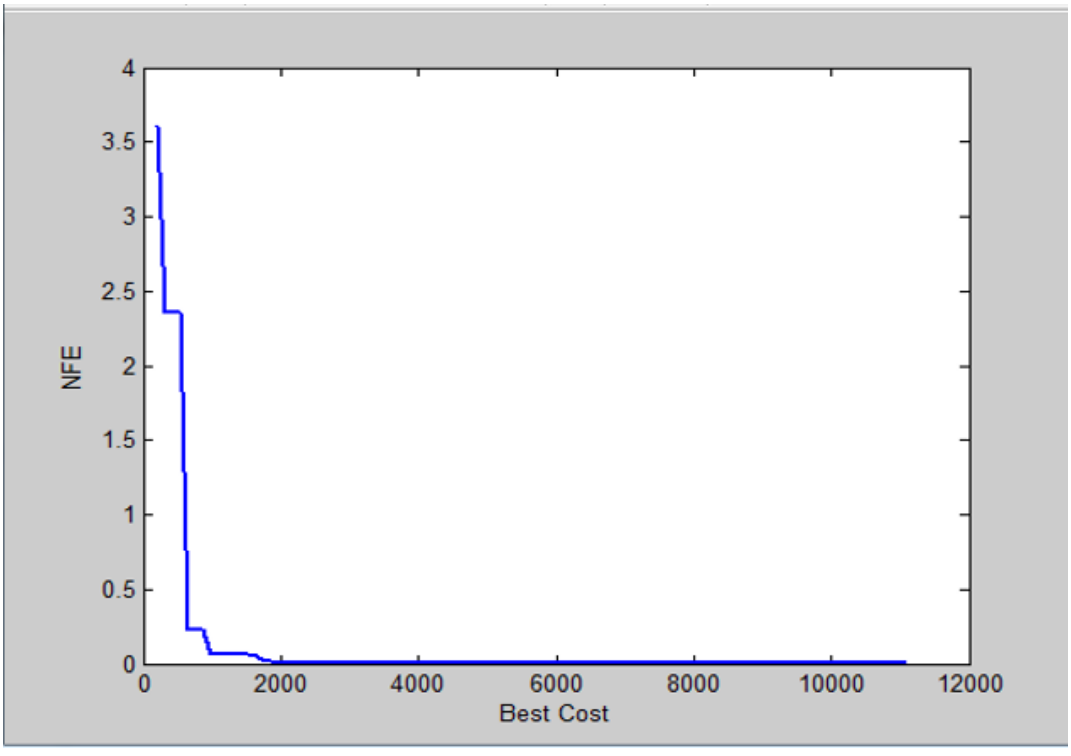


Figure 4. Optimal cost-time diagram with genetics

In this chart, all models are considered as a single model. Additionally, a constant value of 2000 was defined for the system. Under these conditions, the best time is 3.5, and the best cost ranges between 11,000 and

12,000. The final result obtained from the system is represented in the following chart.

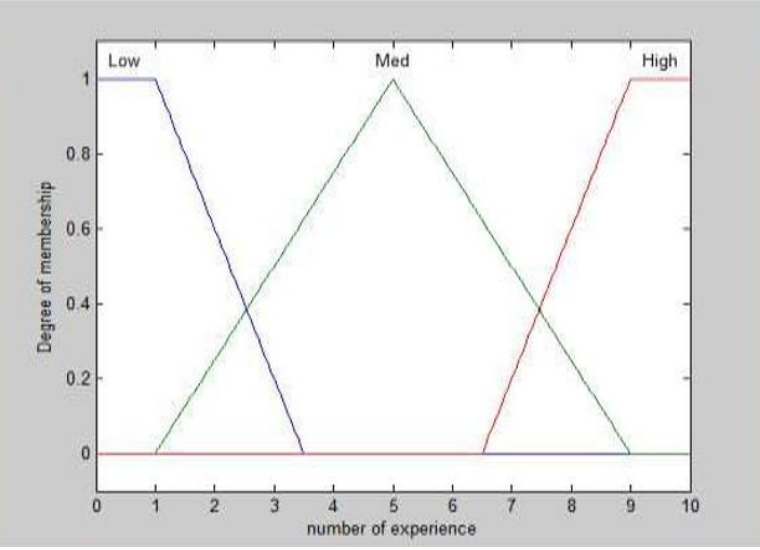


Figure 5. Output diagram of system models.

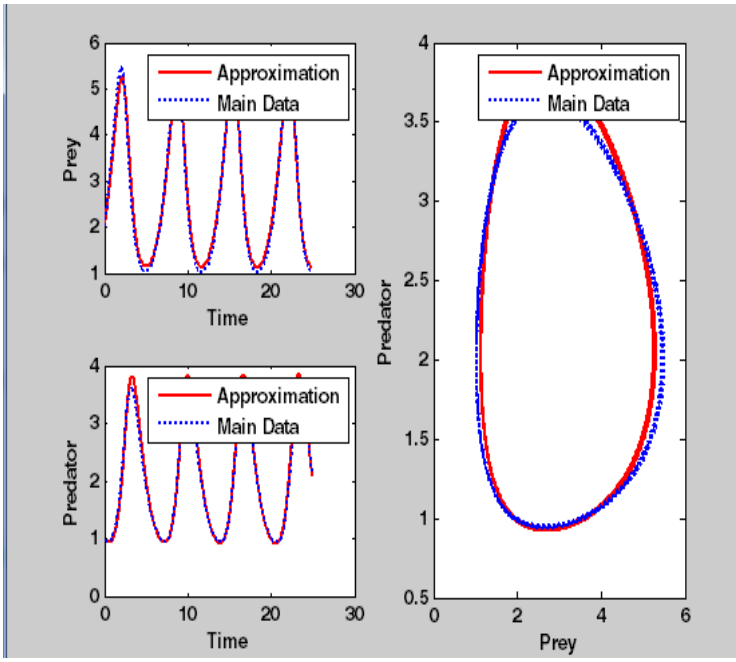


Figure 6. Prioritizing models based on cost and time achieved.

The system operates with some deficiencies compared to the ideal scenario due to the omission of certain minor parameters. To optimize the DGRNN model, various parameters were tested: Optimizer: SGD, ADAM, and ADAMW algorithms were evaluated. Number of Epochs: A range of 1–50 was tested, with early stopping applied upon reaching the highest accuracy to avoid overfitting. Spread Hyperparameter (Sigma): Values of 0.1, 0.2, and 0.3 were tested to minimize the Mean Absolute Error (MAE). Dropout Rate: Rates of 0.1, 0.25, and 0.5 were explored. Training used 70% of the data, with 30% for validation. Loss curves showed overfitting after 40 epochs, so the number of epochs was reduced. The DGRNN

model demonstrated higher efficiency and shorter training times compared to GRNN and other algorithms, with the spread (sigma) parameter being the primary focus for adjustment. Overall, the system in this state consistently outperforms non-fuzzy approaches.

Table 3. Optimal time and cost.

Module	**Optimal time**	**Optimal cost**
M2	8.45	211.42
M4	6.76	156.45
M5	5.00	105.25
M3	5.30	76.11
M1	4.60	32.13

Table 4. Maximum time and cost.

Module	**Maximum time**	**Maximum cost**
M2	10.14	264.27
M4	7.97	190.86
M5	5.75	121.56
M3	3.65	83.72
M1	4.60	33.73

4- Evaluations and Discussion

4.1 Comparison of the proposed method

The proposed method has been compared with other standard methods.

Table 5. Comparison of cost and optimal time of fuzzy output with other methods.

Algorithm	Optimal Cost	Optimal Time
Genetic with KLOC method	12	3.5
Multi-criteria with TPA	18.75	15.44
Fuzzy	23.36	12.03
Evolutionary Algorithm	27.4	15.01

Table 6. Comparison of research work with reference [16].

Row	**Current Research**	**Research [16]**
1	Use of genetic algorithm	Use of fuzzy criteria method
2	Cost and time reduction using the KLOC method	Cost reduction using module prioritization

3	Use of predefined models in the program as a matrix	No use of predefined models in the research (random selection)
4	Use of fixed parameters (KLOC and GA functions) to extract optimal cost and time	Use of optional parameters (TPA, KLOC) to extract optimal cost
5	To optimally improve cost and time, genetic algorithm was initially used as a fixed parameter and then enhanced with the KLOC method	No application of this approach in the program

Table 7. Comparison of the proposed method with other methods.

Method	Time Reduction	Cost Reduction	Error Reduction	Result
Genetic	☐	☐	☐	This method is optimal in terms of cost, time, and low error percentage.
Fuzzy	☐	☐	☐	This method is suitable in terms of cost and time but has a high error rate.
PSO[1]	☐	☐	☐	Comparable in terms of optimization.
BCO[2]	☐	☐	☐	Comparable in terms of optimization.
ACO[3]	☐	☐	☐	Not good in all aspects, but close to optimal.
DE[4]	☐	☐	☐	Consumes higher time and cost compared to other methods.

In the conducted research, we utilized the genetic algorithm for software testing. The work demonstrated superiority compared to other methods in several aspects. Firstly, by prioritizing modules from the lowest to the highest priority, we effectively reduced errors in software testing. Secondly, by employing fixed parameters in the program instead of optional parameters, we minimized the testing time compared to other methods. Lastly, to reduce software testing costs, we used the KLOC method along with the genetic

[1] Particle Swarm Optimization
[2] Bee Colony Optimization
[3] Ant Colony Optimization
[4] Differential evolution

algorithm, repeating this process 2,000 times in the program to achieve optimal cost and time in software testing.

5. Results Analysis

In this study, the results of the proposed method were analyzed from two main perspectives: accuracy and reliability. The evaluation was performed using several key metrics, including Success Agreement (SA) and Effect Size (ES), to measure the reliability of the proposed model in comparison to random guessing. The analysis was

conducted on 28 datasets, with each model being evaluated for its performance across all datasets. The findings presented here aim to provide a detailed comparison of the performance of the proposed method against other machine learning models.

Evaluation Metrics

Success Agreement (SA) measures the level of agreement between the predicted and actual outcomes. An SA value greater than 0.5 indicates that the model's predictions are significantly better than random guessing.

Effect Size (ES) quantifies the magnitude of improvement over random guessing, with higher ES values indicating larger improvements.

Normalized Mean Absolute Error (MAE) was also used as a key performance metric, which allows for the comparison of prediction accuracy across different models. A lower MAE indicates better predictive accuracy.

Performance of the Proposed Model

From Table 3, we observe that the performance of the models varies, but overall, all models, except for Multiple Linear Regression (MLR), outperform random guessing as validated by the SA and ES metrics. Specifically:

Six out of nine models achieved SA values of 0.5 or higher, which means their predictions were significantly better than random guessing.

The ES values for most models hovered around 0.25 or lower, indicating that while the improvements were meaningful, they were relatively modest. This suggests that while the models are effective, there is still room for further refinement to achieve substantial gains in performance.

In terms of accuracy, Random Forest (RF) and Extreme Gradient Boosting (XGB) were the main competitors to the DGRNN model. The DGRNN model consistently delivered better accuracy than the other machine learning models, particularly outperforming the traditional GRNN model. The MAE values clearly indicated that DGRNN provided more accurate predictions across all datasets, showcasing its superiority in defect density prediction.

However, models like MLR and Multi-Layer Perceptron (MLP) ranked as the weakest performers, with poor average MAE values. These results raised questions about the ability of these traditional machine learning models to handle complex datasets, particularly when dealing with sparse output variables and biased predictions. This highlighted the limitations of conventional approaches like MLR and MLP in scenarios where more advanced models like DGRNN are needed.

Distribution of MAE

The normalized MAE results were further analyzed by examining how MAE is distributed across all datasets. A range plot was created to visualize the distribution of MAE values, with a 95% confidence interval (CI). The purpose of this plot was twofold:

To graphically inspect differences among multiple models and draw preliminary conclusions.

To combine the mean, standard deviation, and sample size to generate confidence intervals that reflect a 95% confidence level, which allowed for a more reliable comparison of performance.

From the range plot in Figure 6, it became evident that DGRNN outperformed most of the models, showing significantly better MAE distribution compared to eight out of the nine models. However, models such as ADA, RF, and XGB also showed relatively strong performance. Notably, MLR demonstrated high variability in MAE distribution, indicating that it was not suitable for defect density prediction. KNN and GRNN exhibited similar distributions, showing moderate predictive accuracy. Additionally, models like SVM and ZIFR negatively impacted defect density prediction performance, despite having the smallest confidence intervals, suggesting that they may be ill-suited for this type of task.

Improvement Comparison

In Table 4, a direct comparison was made between DGRNN and other defect density prediction models, using Wilcoxon signed-rank test results for the MAE distributions. The DGRNN model showed significant improvements over various models:

DGRNN vs. GRNN: A notable improvement of 0.53 in MAE.

DGRNN vs. KNN: A significant improvement of 0.55.

DGRNN vs. MLR: A marked improvement of 0.79.

DGRNN vs. RF and XGB: A more modest improvement of 0.25.

These improvements were supported by medium to large effect sizes, suggesting that DGRNN offers notable enhancements in defect density prediction. The significance test further confirmed that DGRNN outperforms most of the models, except for RF and XGB, with DGRNN still maintaining a competitive edge over them in terms of mean MAE.

Impact of Data Dispersion Levels

Another key aspect of the study was the examination of data dispersion levels and their impact on defect density prediction. The datasets used in the study contained a small number of defective modules, with most records having zero defect density. This resulted in sparse data, where modules with zero defect density were considered the majority and the remaining modules with defect density were the minority.

The dispersion ratio (SR) was calculated to measure the level of data dispersion, with higher values indicating greater dispersion. The MAE values were analyzed under different SR levels, as shown in Figures 5 and 6. The shaded regions in the figures represent the 95% confidence interval for the smoothed data (using Loess smoothing).

The results revealed significant variability in the performance of the models, particularly at certain SR levels: At SR levels above 80%, the MAE increased significantly, indicating that predictive performance deteriorated as data dispersion increased.

Classifiers showed better-than-random performance at moderate SR levels, but their performance suffered at very high or very low SR levels.

The findings suggest a negative relationship between performance and SR, with high data dispersion significantly affecting the prediction performance. Therefore, managing data dispersion is critical for improving the accuracy of defect density prediction.

6. Conclusion

The analysis of results highlights the effectiveness and limitations of various predictive models in defect density prediction. The DGRNN model demonstrated superior performance compared to traditional machine learning approaches such as MLR and GRNN, achieving significantly lower MAE values across diverse datasets. While ensemble models like RF and XGB showed competitive accuracy, DGRNN consistently delivered improvements, as evidenced by medium to large effect sizes and significant results in the Wilcoxon signed-rank test. A key insight from the study is the influence of data dispersion on model performance. High levels of dispersion negatively impacted predictive accuracy across all models, including DGRNN, underscoring the challenge posed by sparse datasets with an uneven distribution of defect densities. This finding emphasizes the importance of incorporating techniques to manage dispersion and improve robustness in defect density prediction. The exploration of metaheuristic methods like PSO, ACO, and BCO provided a broader perspective on optimization approaches. While these methods offer advantages in specific scenarios, their time and computational costs often outweigh their benefits for defect density prediction tasks. Similarly, the fuzzy approach, despite being cost- and time-efficient, struggles with high error rates, making it less suitable for accurate predictions. Moving forward, future directions could focus on further refining the DGRNN model to enhance its performance under high dispersion levels, incorporating techniques for better handling of sparse datasets, and exploring hybrid models that combine the strengths of both traditional and metaheuristic approaches. Additionally, investigating the integration of more advanced optimization algorithms and dynamic parameter adjustments could improve the overall accuracy and efficiency of defect

density prediction models.

Conflicts Of Interest:

The author declares that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Funding:

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References:

[1] H. U. Khan, F. Ali, and S. Nazir, "Systematic analysis of software development in cloud computing perceptions," Journal of Software: Evolution and Process, vol. 36, no. 2, p. e2485, 2024. https://doi.org/10.1002/smr.2485

[2] R. Mukherjee and K. S. Patnaik, "A survey on different approaches for software test case prioritization,"

[3] Journal of King Saud University-Computer and Information Sciences, vol. 33, no. 9, pp. 1041–1054, 2021. https://doi.org/10.1016/j.jksuci.2018.09.005

[4] P. Nama, N. H. S. Meka, and N. S. Pattanayak, "Leveraging machine learning for intelligent test automation: Enhancing efficiency and accuracy in software testing," International Journal of Science and Research Archive, vol. 3, no. 01, pp. 152–162, 2021. https://doi.org/10.30574/ijsra.2021.3.1.0027

[5] Y. Wang and S. H. Chung, "Artificial intelligence in safety-critical systems: a systematic review," Industrial Management & Data Systems, vol. 122, no. 2, pp. 442–470, 2022. https://doi.org/10.1108/IMDS-07-2021-0419

[6] M. Hasnain, M. F. Pasha, I. Ghani, and S. R. Jeong, "Functional Requirement-Based Test Case Prioritization in Regression Testing: A Systematic Literature Review," SN Computer Science, vol. 2, no. 6, p. 421, 2021. https://doi.org/10.1007/s42979-021-00821-3

[7] G. Aguiar, B. Krawczyk, and A. Cano, "A survey on learning from imbalanced data streams: taxonomy, challenges, empirical study, and reproducible experimental framework," Machine Learning, vol. 113, no. 7, pp. 4165–4243, 2024. https://doi.org/10.1007/s10994-023-06353-6

[8] V. Etemadi, O. Bushehrian, and G. Robles, "Task assignment to counter the effect of developer turnover in software maintenance: A knowledge diffusion model," Information and Software Technology, Article 106786, 2021. https://doi.org/10.1016/j.infsof.2021.106786

[9] S. Elbaum and M. Hardojo, "An empirical study of profiling strategies for released software and their impact on testing activities," in Proceedings of the International Symposium on Software Testing and Analysis, pp. 65–75, 2014. https://doi.org/10.1145/1007512.1007522

[10] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," ACM Transactions on Software Engineering and Methodology, vol. 6, no. 2, pp. 173–210, Apr. 1997. https://doi.org/10.1145/248233.248262

[11] D. Verma and S. Kumar, "An improved approach for reduction of defect density using optimal module sizes," Advances in Software Engineering, vol. 2014, no. 1, p. 803530, 2014. https://doi.org/10.1155/2014/803530

[12] J. Anderson, M. Azizi, S. Salem, and H. Do, "On the use of usage patterns from telemetry data for test case prioritization," Information and Software Technology, vol. 113, pp. 110–130, 2019. https://doi.org/10.1016/j.infsof.2019.05.008

[13]M. Hasnain, M. F. Pasha, I. Ghani, and S. R. Jeong, "Functional Requirement-Based Test Case Prioritization in Regression Testing: A Systematic Literature Review," SN Computer Science, vol. 2, no. 6, pp. 1–32, 2021. https://doi.org/10.1007/s42979-021-00821-3

[14] Y. Shao, B. Liu, S. Wang, and P. Xiao, "A novel test case prioritization method based on problems of numerical software code statement defect prediction," Eksploatacja i Niezawodność, vol. 22, no. 3, 2020. http://dx.doi.org/10.17531/ein.2020.3.4

[15] V. Etemadi, O. Bushehrian, and G. Robles, "Task assignment to counter the effect of developer turnover in software maintenance: A knowledge diffusion model," Information and Software Technology, Article 106786, 2021. https://doi.org/10.1016/j.infsof.2021.106786