



k-Label Dependent Evolutionary Distance Weighting for Software
Defect Prediction

Muhammad Imran Bin Herman
MSc by Research Computer Science

January 2025

Acknowledgements

I would like to express my deepest gratitude to my supervisors, Professor Rachel Harrison and Dr Hashem Dehghanniri, whose unwavering support, guidance, and encouragement have been invaluable throughout this research journey. Their expertise and insightful feedback have not only enhanced the quality of this work but also inspired me to explore new perspectives. It has been an honor and a privilege to work under your guidance. I am truly grateful for the time, effort, and belief you have invested in me. This thesis would not have been possible without your support and encouragement.

I would also like to thank Dr Daniel Rodriguez, his willingness for sharing his expertise and experience has been instrumental in broadening my knowledge, which was useful to my academics life. His guidance has not only contributed significantly to this research but has also greatly enriched my academic journey.

I would also like to express my appreciation to my family who always shown their support and motivation for pursuing my studies. Bless my parents for helping me throughout my entire study years. In addition, special thanks to my friends, who were always there for me through thick and thin throughout my studies. Their moral support is the most powerful motivator for me to overcome any issues or barriers in my education and life.

Abstract

Software Defect Prediction is a field of study that uses machine learning algorithms to identify software that is susceptible to defects. Software defects and bugs can potentially cause errors that have significant consequences ranging from minor inconveniences to system failures. Identification of errors at an early stage allows software developers to reduce resources used and to provide more reliable software. Conventional SDP models often depend on static feature-weighting approaches that cannot be adjusted to the particularities of different data sets. Existing models address some of these challenges, but lack interpretability or prediction accuracy. This research addresses the limitations of existing models by introducing an enhanced k -Label-Dependent Evolutionary Distance Weighting (k -LDEDW) method, which combines an adaptive genetic algorithm and k -NN classifier (k -NN) to improve predictions by dynamically adjusting the feature weights for each data point. The proposed adaptive k -LDEDW model is implemented to predict software defects and the performance of the model is evaluated in the ApacheJIT data set, which contains unbalanced commits of clean and bug-inducing software. The adaptive genetic algorithm refines the weights using crossover and mutation techniques, guided by Matthew correlation coefficient (MCC) as the fitness function. In addition, Shapley values are used to explain the importance of software features in defect predictions to provide insights into the interpretability of the model. The k -LDEDW model is then compared with graph neural networks (GNN), clustering techniques, and JIT bug prediction models. The respective models are compared using evaluation metrics for accuracy, precision, recall, f_1 -score, AUC, and MCC. The results show that k -LDEDW outperformed other methods, achieving an MCC of 0.65, precision of 0.86, recall of 0.98, and an f_1 -score of 0.92 for clean commits. For buggy commits, it attained a precision of 0.91, while the overall predictions yielded a precision of 0.88 and recall of 0.76. Shapley values revealed that k -LDEDW utilized a diverse range of features, showing that the model offers a holistic evaluation of software features. These results highlight the robustness of k -LDEDW and its ability to adapt to an imbalanced data set.

Contents

1	Introduction	4
1.1	Overview	4
1.2	Motivation	5
1.3	Problem Statement	5
1.4	Aim and Objectives	6
1.5	Scope	6
1.6	Significance of Study	6
1.7	Summary	7
2	Literature Review	8
2.1	Overview	8
2.2	Evolution of Methods in SDP	9
2.2.1	Machine Learning Methods in SDP	9
2.2.2	Deep Learning in SDP	11
2.3	k -Nearest Neighbors	12
2.3.1	Training Data	13
2.3.2	Distance Metrics	13
2.3.3	k -NN Classification	15
2.4	Genetic Algorithm	16
2.4.1	Initialization and Encoding Schemes	16
2.4.2	Selection Mechanisms	17
2.4.3	Crossover Operators	18
2.4.4	Mutation Operators	19
2.4.5	Existing Studies utilizing GA optimization	21
2.5	Summary of Reviewed Methods	22
2.6	Utilization of GA for k -NN optimization	23
2.7	Justification of proposed approach	24

3	Methodology	25
3.1	Overview	25
3.2	Research Method	25
3.3	Problem Identification	26
3.4	Analysis of Existing Models	28
3.5	Design of Proposed approach	29
3.5.1	Input Data	29
3.5.2	Data Pre-processing	29
3.5.3	Initialization of Weighting Matrices	30
3.5.4	Genetic Algorithm to Optimize Weights	30
3.5.5	Genetic Operators	30
3.5.6	Blend Crossover (BLX- α)	31
3.5.7	Mutation	32
3.5.8	Adaptive Mutation and Crossover Rates	32
3.5.9	Integration of Adaptive Rates	32
3.5.10	Fitness function	33
3.5.11	Generational Replacement	34
3.5.12	Iteration Process of Genetic Algorithm	35
3.6	Implementation	36
3.6.1	Data	36
3.6.2	Random Sampling Method	38
3.7	Model Evaluation	39
3.7.1	Evaluation Metrics	39
3.7.2	Shapley Values	39
3.7.3	Understanding Shapley values	40
3.7.4	Model Comparison	41
3.7.5	Temporal Cross-Validation for Just-In-Time Defect Prediction	41
4	Implementation and Results	42
4.1	Overview	42
4.2	Adaptive Mutation and Crossover Rates	43
4.2.1	Adaptive Crossover Rate	43
4.2.2	Adaptive Mutation Rate	43
4.2.3	Example of Adaptive Rates	44
4.3	Multiple Mutation Implementation	45
4.3.1	Uniform Mutation	45
4.3.2	Gaussian Mutation	45

4.3.3	Swap Mutation	46
4.4	Results of Adaptive k -LDEDW Implementation	48
4.4.1	k -LDEDW Model Performance	48
4.4.2	Performance Analysis Across Different k -values	50
4.4.3	Evaluation under Temporal Cross-Validation	51
4.4.4	Convergence Analysis of Genetic Algorithm and Weight Stability	51
5	Evaluation & Discussion	54
5.1	Model Comparison	54
5.1.1	Comparison with GNN models	54
5.1.2	Clustering techniques and JIT bug prediction techniques	56
5.2	Model Explainability	56
5.2.1	Shapley values for k -LDEDW	56
5.2.2	Shapley values for k -NN	57
5.2.3	Comparison of Shapley Values	58
5.3	Significance of Results in Software Defect Prediction	59
6	Threats to Validity	60
6.1	Internal Validity	60
6.2	External Validity	60
6.3	Construct Validity	61
7	Conclusion & Future Work	62
7.1	Conclusion	62
7.2	Mapping Research Questions to Thesis Sections	64
7.3	Contribution	65
7.4	Future Work	65

Chapter 1

Introduction

1.1 Overview

The field of Software Defect Prediction (SDP) leverages a wide range of machine learning algorithms to identify softwares that are prone to defects. However, the intricate nature of software development causes this area of research to be prone to limitations (Pachouly et al., 2022; Thota et al., 2020). Software errors can cause significant consequences, ranging from minor inconveniences to system failures (Prakash et al., 2015). The field of SDP emerges as a critical defense against such risks. By proactively identifying potential errors, developers are aware of potential trouble spots, which leads to reduced resources required to provide robust and reliable software (Pachouly et al., 2022). Although existing models have demonstrated outstanding capabilities, there remains room for improvement (Šikić et al., 2021). Thus, it remains imperative to research further in this field.

Weight-based models have been extensively explored in machine learning, specifically algorithms for classification tasks (Mateos-García et al., 2012). The adjustment of weights during the iterative process of the training phase allows the model to better understand the data set. Neural network models are one of the most prominent examples (Fernando and Tsokos, 2021; Li et al., 2012; Park, 2009), along with support vector machines (Goyal, 2022) and nearest-neighbors (Iqbal et al., 2019) to utilize weights for a more accurate fit. All the weighting proposals have the goal of finding the optimal values whilst attempting to avoid over-fitting and under-fitting. A few ways to carry out optimization is by using analytical methods such as gradient descent (Ruder, 2016) and least squares estimation (Shi et al., 2023), and heuristics such as evolutionary algorithms such as Particle Swarm Optimization (PSO) (O’Neill et al., 2018).

1.2 Motivation

Existing SDP models utilize various techniques to analyze software characteristics and predict how vulnerable it is to defects and bugs. The exploration of utilizing global weights for all features has been extensive in the past (Esteves et al., 2020; Albattah and Alzahrani, 2024). However, these approaches may overlook the patterns that exist at the individual data point level (Zou et al., 2021). This limitation creates a compelling need for a more flexible and dynamic feature of the weighting mechanism. Several proposed methods are implemented that use weights applied to regions of the data set (Fernández and Isasi, 2008) or class labels (Chen et al., 2011). This research focuses on optimizing with a set of weights for each data point.

The main motivation of this research is to introduce an enhanced k -Label Dependent Evolutionary Distance Weighting (k -LDEDW) approach. k -LDEDW method was implemented in (Mateos-García et al., 2012) that utilize the k -Nearest Neighbors (k -NN) algorithm, along with the weighting matrix. The weighting matrix is optimized using an adaptive genetic algorithm, this will be further explained in Chapters 2 and 3. In addition, the recent availability of the ApacheJIT data set provided by (Keshavarz and Nagappan, 2022), which is a large imbalanced data set with software metrics, also motivates the implementation of k -LDEDW in SDP. The research carried out by (Mullick, 2021) suggesting k -LDEDW algorithm performed well in imbalanced data set further encourages this research.

1.3 Problem Statement

Due to software systems being inherently complex and dynamic, Software Defect Prediction (SDP) is still a difficult undertaking. Conventional SDP models often depend on global weights or static feature weighting approaches that are unable to adjust to the particularities of different data sets (Menzies et al., 2010). Although these models have shown respectable performance, their restricted prediction accuracy and decreased applicability in various software development contexts are caused by their inability to dynamically modify feature importance (Haldar and Capretz, 2024b).

Existing models such as Graph Neural Networks (GNN) (Keshavarz and Rodríguez-Pérez, 2024; Pornprasit and Tantithamthavorn, 2021), clustering techniques, and JIT bug prediction models (Jadhav et al., 2023) address some of these challenges, but often lack interpretability or unable to identify dependencies between features. There is a need for a model that dynamically adjusts feature importance while maintaining interpretability and scalability.

The method known as k -Label Dependent Evolutionary Distance Weighting (k -LDEDW) offers a possible solution to this constraint. This approach makes use of a dynamic weighting matrix, in which a genetic algorithm continuously modifies the relevance of each feature. However, even though k -LDEDW has demonstrated promise in managing unbalanced datasets, its full potential has not yet been fully realized, especially when working with intricate datasets like ApacheJIT software repositories. Therefore, to offer a more focused approach to understanding the k -LDEDW model, the following research questions are proposed.

1. a. To what extent does the k -LDEDW method perform on the ApacheJIT dataset?
b. How well does the k -LDEDW method distinguish between clean and buggy commits in the ApacheJIT dataset using standard evaluation metrics?
2. How well does the k -LDEDW performs compared to existing software defect prediction techniques?
3. To what extent do Shapley values provide interpretability of the feature-specific contributions of the k -LDEDW model in predicting software defects?

1.4 Aim and Objectives

The aim of this research is to develop and validate a software defect prediction model by integrating evolutionary algorithms with the k -Nearest Neighbors (k -NN) method. The following are the sub-objectives of the research:

1. To propose an evolutionary optimization method for k -NN (k -LDEDW) in Software Defect Prediction.
2. To implement and evaluate the performance of the k -LDEDW method using ApacheJIT data set using multiple evaluation metrics.
3. To Compare the proposed method with existing Software Defect Prediction Techniques such as Graph Neural Networks (GNN), clustering techniques, and JIT bug prediction techniques.
4. To investigate k -LDEDW's model interpretability using Shapley values.

1.5 Scope

This research focuses on the development, evaluation and comparison of the k -LDEDW approach for SDP. This encompasses designing the k -LDEDW algorithm, incorporating a dynamic weighting system that prioritizes software features based on similarity between data points and utilizing the k -NN model for classification. Also, using the Just-in-Time dataset (Keshavarz and Nagappan, 2022) that contains software code metrics and defects information. Pre-processing the data set to ensure suitability for k -LDEDW algorithm. Afterwards, two studies utilizing the same data set will be selected as benchmarks for comparison based on evaluation metrics.

1.6 Significance of Study

Accurate predictions, along with model interpretability are a fundamental aspect in SDP. Mainly, improved software quality as early detection of defects allows for a timely rectification, thus leading to a more robust and reliable software. The proposed method (k -LDEDW) offers a possible method to provide accurate

prediction and model interpretability. In addition to that, by pinpointing defect-prone areas, developers can focus their efforts on these critical areas, reducing workload, and saving valuable time and resources. This can be seen in a Cambridge study that reports that the cost of software defects increased by 312 billion\$ per year (Brady, 2013) while (Tassey, 2002) states that the national annual costs of an inadequate infrastructure for software testing is estimated to range from 22.2 to 59.5 billion. Hence, having a good software prediction model can lead to a more efficient process of Validation and Verification (V&V).

1.7 Summary

This chapter has provided an outline of the purpose of this research, the research questions being addressed, objectives outlined, significance of the study, and the scope of this research. The following chapter will go deeper into the existing literature on this area of study. Chapter 3 will highlight the technical details of the k -LDEDW algorithm, the research methodology. Chapter 4 provides and in depth analysis of results obtained from this research. Followed by chapters, threats to validity and conclusion.

Chapter 2

Literature Review

2.1 Overview

Chapter 1 provided a detailed introduction to the research, focusing on the critical task of detecting software defects, particularly in weight-based models. The chapter addressed the motivations for this research, emphasizing the importance of accurate prediction of bug-induced in software engineering. The challenges associated with software defect prediction models were outlined, including the complexity and variability of unbalanced data sets. The problem statements identified the limitations of existing models, such as model interpretability establishes the need for an advanced weight-based interpretable model. The research questions are formulated to provide guide and focus for the investigation of the k -LDEDW model, and the objectives were set to evaluate the model, which integrates an adaptive genetic algorithm with the k -NN classification algorithm.

Chapter 2 serves to provide a comprehensive background to the research, establishing theoretical foundations and historical context that underpin this research. In the following section 2.2 explains the evolution of methods used in this literature including machine learning and deep learning algorithms. In Section 2.3, an in-depth review of k -NN algorithm is provided. Section 2.4 provide a detailed explanation of the genetic algorithm model, reviewing existing operators and research implementations in the literature. In Section 2.5, a summary of models reviewed in the literature is provided. The following section 2.6 provides the existing implementations of Genetic Algorithm for k -NN. Finally, Section 2.7 provides justifications for the proposed model that is implemented in this research.

2.2 Evolution of Methods in SDP

The field of software defect prediction has seen significant transformation over the years, beginning from statistical methods to machine learning and on to deep learning revolution and current innovative trends. This evolution of methods has been for the same purpose, more efficient, and accurate prediction to analyze the ever-increasing size of data in this field. Advancements in this field facilitate various applications for consumers, ranging from mobile applications and gaming platforms to critical systems such as healthcare and aerospace.

The rise of large-scale software systems, particularly in large and complex applications, has provided support to significant advancements in software defect prediction. At the same time, the field of machine learning, which is a subset of Artificial Intelligence (AI), has become accessible to the general public, including researchers and software developers. The applications of software defect prediction have shown growth in various fields of software quality assurance. AI encompasses a huge domain of computer science focused on creating systems that are capable of performing tasks traditionally requiring human reasoning. This is achieved through simulations of human cognitive processes to analyze and interpret complex data (Jadhav and Barati Farimani, 2022). Recent methods utilizing AI includes machine learning and deep learning, which has been shown to transform the landscape of software defect prediction. These advancements allow systems to make accurate predictions about software defects, matching or even surpassing human expertise without requiring explicit programming for each specific scenario (Bowling et al., 2006).

2.2.1 Machine Learning Methods in SDP

Machine Learning (ML) has been an essential instrument to address the limitations of statistical approaches in SDP. Early ML models such as k -Nearest Neighbors (k -NN) and Decision Trees (DT) have shown the potential for automated and efficient prediction. These models tackled challenges where statistical methods are limited: handling imbalanced data sets, high-dimensional data, and complex feature relationships.

Decision Trees (DT) algorithms, such as C4.5 (Quinlan, 1993), are among the earliest ML methods in SDP. They partition datasets recursively based on feature thresholds, constructing a tree structure that represents decision rules and class predictions. The flexibility of DTs to handle imbalanced datasets was highlighted by (Khoshgoftaar et al., 1998). The DT method poses a potential drawback from overfitting, particularly when grown to full depth. This limitation can be addressed by pruning and implementing ensemble methods like Random Forests (RF). Recent studies, such as (Chinenye et al., 2023), demonstrate how DTs integrated with ensemble techniques improve prediction accuracy.

The simplicity and adaptability of k -Nearest Neighbors (k -NN) make it a foundational algorithm for SDP. It classifies software modules based on the labels of their k nearest neighbors, adapting to various data distributions without assuming underlying statistical models (Khoshgoftaar et al., 2002). Enhancements like weighted k -NN (Zhang and Zhou, 2005) and feature selection methods (Hammad et al., 2019) address its

sensitivity to feature scaling and the choice of k . Despite its strengths, k -NN’s performance is limited by the curse of dimensionality and computational inefficiency in large datasets (Schuh et al., 2014).

Using kernel functions, Support Vector Machines (SVM) are efficient at processing high-dimensional data and modeling non-linear relationships (Kulamala et al., 2021). Research like (Gray et al., 2010) shows how well SVMs predict outcomes in SDP problems. However, with large datasets, their computational intensity during training presents difficulties. To increase accuracy and efficiency, variations have been created, such as Kernel Principal Component Analysis (KPCA-SVM) (Zhou et al., 2019) and LASSO-SVM (Wang et al., 2021b).

Naive Bayes (NB) is a model that offers simplicity and computational efficiency by leveraging Bayes’ theorem. The ease of implementing this method allows it to be implemented in SDP (Jiang et al., 2008). Although its independence assumption can limit the effectiveness when it comes to output prediction, enhancement like Feature-Dependent Naive Bayes (FDNB) (Ömer Faruk Arar and Ayan, 2017) and Weighted Naive Bayes (WNB) (Ji et al., 2019) addressed these challenges, leading to an improved defect prediction.

SDP has been transformed by ensemble techniques like Random Forests, AdaBoost, and Gradient Boosting. Boosting strategies repeatedly focus on misclassified examples to improve model performance (Beja-Battais, 2023), while Random Forests integrate several decision trees to prevent overfitting (Lessmann et al., 2008). Bagging and boosting are used in hybrid ensemble techniques, such as those in (Khan, 2020), to optimize the prediction accuracy. Nevertheless, these techniques are computationally demanding and frequently require significant resources.

Clustering techniques, such as k -means and hierarchical clustering, are frequently used in SDP to group software modules based on feature similarity. These methods are particularly effective in preprocessing, helping to identify patterns in unlabeled data and improving the performance of supervised learning models (Alsaeedi and Khan, 2019). For example, k -means clustering has been applied to segment defect-prone and clean modules, enabling more targeted classification by supervised algorithms (Gan et al., 2018). Hierarchical clustering, when combined with dimensionality reduction methods like Principal Component Analysis (PCA), has demonstrated success in reducing noise and highlighting critical patterns in software metrics (Khalid et al., 2023). Despite their advantages, clustering methods depend heavily on selecting an appropriate number of clusters, which requires careful tuning, and they are susceptible to being influenced by outliers (Oyewole and Thopil, 2023). It is worth mentioning that (Jadhav et al., 2023) implements clustering techniques and JIT bug prediction techniques on the ApacheJIT data set, which is the data set used in this research. An overview of the data set will be provided in section 3.6.1.

Unsupervised machine learning algorithms, such as clustering methods, namely hierarchical clustering and k -means are commonly used in SDP to group software modules according to feature similarity. The performance of supervised learning models is enhanced by these techniques, which are especially useful for preprocessing and assisting in the discovery of patterns in unlabeled data (Alsaeedi and Khan, 2019). For instance, k -means clustering has been used to separate clean and defect-prone modules, allowing supervised

algorithms to classify data more precisely (Gan et al., 2018). It has been shown that hierarchical clustering may effectively reduce noise and reveal important trends in software metrics when paired with dimensionality reduction techniques like Principal Component Analysis (PCA) (Khalid et al., 2023). Despite their benefits, clustering techniques are vulnerable to outlier effect and rely significantly on choosing the right number of clusters, which necessitates careful tuning (Oyewole and Thopil, 2023).

Machine learning has greatly improved this field by tackling prominent issues in SDP. This include unbalanced data sets, high-dimensional data, and intricate feature interactions. While techniques such as SVM and NB added the robustness and scalability to prediction tasks, algorithms such as k -NN and DT established the foundation for effective interpretable models. Through their integrated learning approaches, ensemble methods further improved accuracy and reliability, while clustering techniques provided insightful information for pattern recognition and preprocessing. When combined, these techniques offer a broad range of tools to address the intricacies of SDP, facilitating automatic and more precise defect identification across various software systems.

2.2.2 Deep Learning in SDP

Deep learning architectures offer a way to model complex relationships in software metrics when automating feature extraction. Important designs include artificial neural networks (ANN), convolutional neural networks (CNN), recurrent neural networks (RNN), deep neural networks (DNN), and graph neural networks (GNN).

Artificial Neural Networks is among the first neural network models applied in this field (Khoshgoftaar and Szabo, 1996). ANNs consist of layers of interconnected neurons, in which excel at studying nonlinear relationships (Khoshgoftaar et al., 1997). Dropout layers (Srivastava, 2013) and synthetic oversampling (Dipa and Sunindyo, 2021) are used in recent studies to decrease overfitting and enhance performance on imbalanced datasets. Although it is a simple model, ANNs require a lot of parameter tweaking, which may be computationally demanding (Panda, 2019).

Deep Neural Networks (DNN) allow ANNs to learn hierarchical feature representations by extending a number of additional hidden layers (He et al., 2016). In complex SDP tasks, studies such as (Manjula and Florence, 2019) demonstrate the greater accuracy of DNNs. By tackling problems such as overfitting and high computational costs, strategies like early halting and L_1 normalization (Wu et al., 2018) improve their generalization capabilities.

By converting software metrics into spatial representations, Convolutional Neural Networks (CNN), which were originally created for image processing, have been modified for SDP (Li et al., 2017). According to (Qiu et al., 2022), the model excels in identifying local patterns in modular software systems. However, their application is limited by their dependency on big data sets with labels and difficulties processing irregular data structures.

Recurrent Neural Networks (RNN) and their variant, Long-Short Term Memory Networks (LSTM) are

excellent at representing sequential data, which makes them appropriate for examining temporal trends in software evolution (Liu et al., 2018). RNNs can learn long-term dependencies thanks to LSTMs, which solve the vanishing gradient issue (Lu and Salem, 2017). Their efficiency in defect prediction tasks is demonstrated by studies such as (Wang et al., 2021a), especially when examining the rate of defects and code evolution. However, more sophisticated architectures like LSTMs partially address the vanishing gradient problem, which limits RNNs’ capacity to learn long-term dependencies (Le and Zuidema, 2016).

In contrast to conventional models, GNNs can cope with the linkages between different parts of the software system, and how those connections might lead to defects (Wu et al., 2020a). For instance, a study by (Qiao et al., 2024) employs GNNs to accurately forecast defects in modular systems, while (Cui et al., 2022) showed higher defect detection rates by integrating code dependencies and developer activity. By concentrating on important nodes and edges, attention methods (Veličković et al., 2017) significantly enhanced interpretability and performance. However, challenges like high computational complexity and sensitivity to graph size persist (Wu et al., 2020b). It is worth mentioning that (Keshavarz and Rodríguez-Pérez, 2024) implements a GNN model on the ApacheJIT data set, the data set used in this research.

Software defect prediction (SDP) has greatly improved with the help of deep learning architectures, which makes it possible to automatically extract features and model intricate relationships between software metrics. Each deep learning method provides different improvements, from basic ANNs and hierarchical DNNs, approaching the problem using spatial software metrics (CNN), sequence-focues RNN and LSTM, and graph-based neural network (GNN). These techniques collectively show the revolutionary influence of deep learning on SDP, providing reliable, scalable solutions for detecting and mitigating software faults in more complex systems, even while computing demands and data needs still present difficulties.

2.3 k -Nearest Neighbors

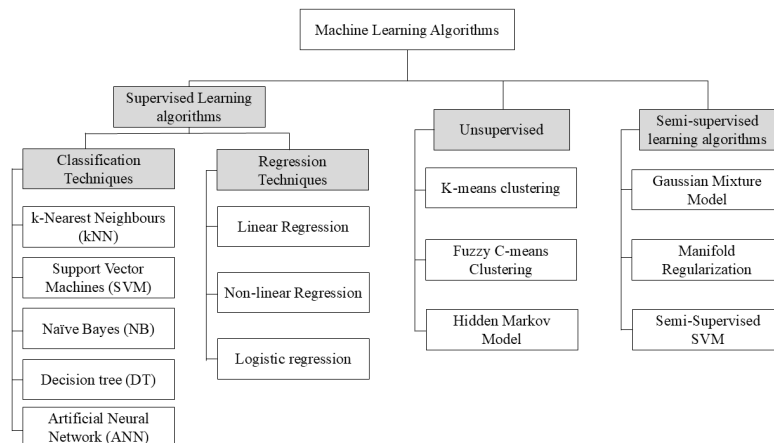


Figure 2.1: Taxonomy of Machine Learning Algorithms adapted from (Messaoud et al., 2020)

k -Nearest Neighbors (k -NN) is a supervised machine learning algorithm that uses the proximity of a data point to a set of data on which it is trained to make predictions. The simplicity of the k -NN model allows users to quickly understand and implement it as it is one of the most simple classification method (Peterson, 2009). In terms of core concept, k -NN operates on the principle that similar data points tend to cluster together (Abu Alfeilat et al., 2019). It is also a versatile and intuitive technique that can be used in various applications (Venkateswarlu and Gangula, 2024), such as classification and regression tasks. The model first learns the data set, then identifies its nearest neighbors based on the distance between them. Consequently, the model makes a prediction based on these neighbors. Despite its simplicity, k -NN has limitations, such as its sensitivity to the curse of dimensionality (Schuh et al., 2014), which can degrade performance as the number of features increases. Addressing these limitations often involves incorporating techniques such as weighting schemes to assign different levels of importance of features for each data point, as explored in this work.

2.3.1 Training Data

The k -NN model initially is trained on a dataset consisting of data points and feature columns. Each data point is fitted into a multi-dimensional space, where each dimension represents a specific feature or attribute of a single data point. Given a dataset, D with N rows and f features, it can be represented as $D = \mathbb{R}^{N \times f}$. The dataset is split into training data, TR and testing data, TS such that $TR \cap TS = \emptyset$ and $TR \cup TS = D$.

The k -LDEDW algorithm introduces a feature-specific weighting matrix for the calculation of k -NN. In the k -LDEDW algorithm, the weighting matrices are tailored to each feature of the data set. This approach assigns a unique set of weights to each feature for every data point, eliminating the use of global weights across features. This dynamic weighting approach adapts the distance metric to the local structure of the data, particularly in data sets with different feature importance. This will be further discussed in Section 3.5.3.

2.3.2 Distance Metrics

After learning about the data, the model utilizes a distance metric to quantify the distance between two distinct data points. The more common distance metrics include the Euclidean distance (Peterson, 2009), Manhattan distance (Malkauthekar, 2013), and Minkowski distance (Merigó and Casanovas, 2011) that can be seen in equation (2.1), (2.2) and (2.3). For a given data set with f features, the distance between two data points x and y can be written as follows

$$d_{Euclidean}(x, y) = \sqrt{\sum_{i=1}^f (x_i - y_i)^2} \quad (2.1)$$

In most cases, distance will refer to Euclidean distance by default. In other words, the Euclidean distance or Euclidean metric is the "ordinary" distance between two points (Goel and Khanna, 2008). It is notable

for its computational simplicity (Goldani, 2024) but is also sensitive to outliers (Nawara and Kashef, 2021). As for the Manhattan distance,

$$d_{Manhattan}(x, y) = \sum_{i=1}^f |x_i - y_i| \quad (2.2)$$

Manhattan distance in our case is less sensitive with respect to atypical observations (Filzmoser et al., 2008). One drawback of Manhattan distance is that it only measures distances along the x and y axes. It doesn't account for diagonal connections or relationships between points (Wang, 2023). The Minkowski distance can be written as follows:

$$d_{Minkowski}(x, y) = \left(\sum_{i=1}^f |x_i - y_i|^\beta \right)^{1/\beta} \quad (2.3)$$

where β is an integer. Minkowski distance adapts its contour shape as the parameter β changes, making it flexible for approximating the solution landscape in single-objective genetic algorithms (Xu et al., 2018). Since this research is focused on using weights, it is also fundamental to include weighted distances for each of the respective equations.

$$d_{Euclidean}(x, y, \omega) = \sqrt{\sum_{i=1}^f \omega_i (x_i - y_i)^2} \quad (2.4)$$

$$d_{Manhattan}(x, y, \omega) = \sum_{i=1}^f \omega_i |x_i - y_i| \quad (2.5)$$

$$d_{Minkowski}(x, y, \omega) = \left(\sum_{i=1}^f \omega_i |x_i - y_i|^\beta \right)^{1/\beta} \quad (2.6)$$

where ω_i is the global weight of the respective columns of the data set. While each distance metric has their advantages and disadvantages, this research implements the weighted distance (Paredes and Vidal, 2006) that follows

$$d_w(x, y, \omega) = \sqrt{\sum_{k=1}^f \omega_{label(x)_k}^2 (x_k - y_k)^2} \quad (2.7)$$

where the $\omega_{label(x)_k}$ is the weight corresponding to row of $x \in TR$, while $y \in TS$. This distance metric represents the optimal distance because the weights depend on the class of the training sample (Mateos-García et al., 2019). This enhances the standard Euclidean distance by incorporating feature-specific weights. This adjustment allows the model to emphasize more relevant features. Consequently, the calculation of the distance between a testing sample, y and a training sample, x is influenced by the "label" of x . It should be noted that label of x in this expression represents the specific weight of training sample x , corresponding to its row and column from D .

2.3.3 k -NN Classification

The model classification first selects the k closest samples (nearest neighbors within proximity) for all the training samples. From the information obtained, it predicts the test sample with a simple classifier. It should be noted that the critical part of this process is to identify the right value for k , which determines the number of neighbors to consider. It has the following impact on the model:

- **Small k :** For smaller k such as $k = 1$, the model is too sensitive to noise and anomalies, which leads to over-fitting. In such situations, classification may be ruled by aberrations or outliers.
- **Large k :** On the other hand, a larger k can smooth the decision boundary, resulting in under-fitting. This is because the algorithm averages information across a large number of points, which may include points from the two classes.

In this research, the method used is the majority classifier. The majority classifier assigns the label of the test sample based on the most frequent label among its k -nearest neighbors. For instance, if there are 2 classes C_1 and C_2 , and the majority of neighbors belong to class C_1 , The test sample is classified as class C_1 . This choice is due to it being the standard prediction approach to k -NN (Zhang, 2021), making it easier to understand the method due to its classification nature. There are a few other alternatives to this classification method, namely:

- **Weighted Voting:** Instead of giving equal weight to all nearest neighbors, this method assigns greater weights to neighbors that are closer to the test sample. This approach accounts for the intuition that closer points are more likely to share the same label as the test sample (Kuncheva and Rodríguez, 2014).
- **Distance Thresholding:** Instead of fixing k , neighbors within a specific distance threshold are considered for classification. This can dynamically adjust the number of neighbors based on the density of the data (Hazra and Santra, 2019; Aguzzi et al., 2009).

2.4 Genetic Algorithm

Genetic Algorithms (GAs) fall under the broad category of Artificial Intelligence (AI), which encompasses all methods that enable machine learning to exhibit intelligent behavior. Within AI, GAs belong to a subset of Computational Intelligence, a field focusing on algorithms that are inspired by natural processes and human reasoning. Within computational intelligence, evolutionary algorithms are a group of optimization techniques modeled after natural selection and evolution. Genetic algorithms lies within the evolutionary algorithm, a specific type that use operations like crossover, mutation and selection to solve optimization problems. The hierarchical diagram can be seen as follows:

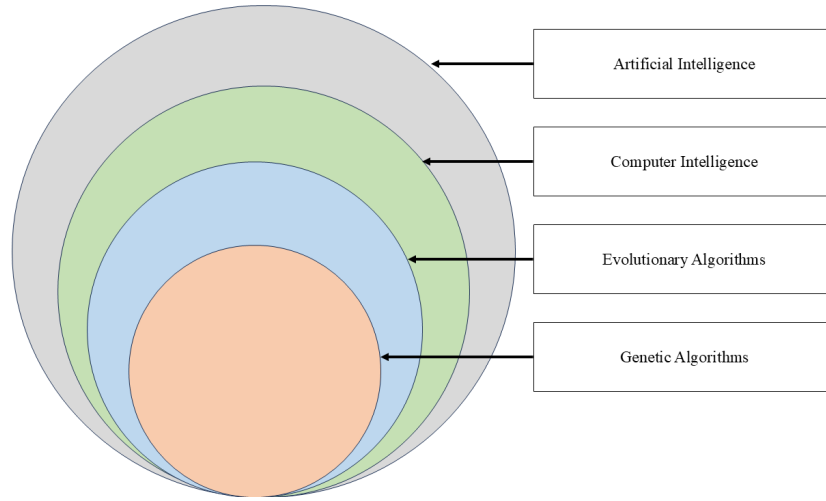


Figure 2.2: The hierarchical flow of Genetic Algorithm adapted from (Yang et al., 2022)

These algorithms emulate the process of biological evolution to find solutions to complex optimization and search problems (Ling, 2023). Since their inception, GAs have been applied across diverse fields, including encryption, image processing, video processing, medical imaging, and gaming due to their versatility and robustness in solving optimization problems (Katoch et al., 2021). This section seeks to provide an in-depth review of GAs, focusing key components such as individual encoding, genetic operators, fitness functions, and rules to carry out the generational replacement. Each components plays a vital role in guiding algorithms to find the optimal solution.

2.4.1 Initialization and Encoding Schemes

The GA process begins with the initialization of a population of candidate solutions. The encoding schemes are differentiated according to the problem domain. It could be binary, octal, hexadecimal, permutation, value-based and tree (Katoch et al., 2021). This depends on the problem domain requirements (Eiben and Smith, 2015).

The most often used encoding system is binary encoding. A string of 1 or 0 is used to represent each gene or chromosome (Sivanandam and Deepa, 2008). Each bit in binary encoding represents the properties of the solution. Crossover and mutation operators can be implemented more quickly thanks to this feature. However, converting data into binary form takes more work, and the binary conversion determines the algorithm's correctness.

In addition to binary encoding, the value encoding scheme represents the individuals using a string of some values that may be characters, actual numbers, or integers (Goldberg et al., 1991). When dealing with situations involving more complex values, this encoding approach may be useful, including Real-Coded GAs (RGA). This is because binary encoding may be implemented in some situations. It is used primarily in neural networks to determine the ideal weights. Tree encoding represents the gene by a tree of functions or commands. This method of individual encoding is very similar to the representation of repression in tree format (Jebari and Madiafi, 2013).

For situations where the solutions are in the form of matrices, an individual can be specially designed to fit this situation. For example, an individual can be a set of numbers represented by a matrix, with rows and columns. This method is implemented in (Mateos-García et al., 2012), and will be used in this research paper.

2.4.2 Selection Mechanisms

The selection mechanism is a very important step in genetic algorithms. It determines whether the particular individual will participate in the next generation or not. The convergence rate of GA depends on the selection pressure (Yadav and Sohal, 2017). The well-known methods include the roulette wheel, rank, and tournament selection method.

Roulette wheel selection method is one of the traditional GA selection technique. This is due to its simplicity, all of the individuals are fitted on a roulette wheel, according to their fitness value. A segment of the wheel is assigned to each individual whose size is proportional to the fitness value of the individual (Razali and Geraghty, 2011). This means that a better fitness value provides a larger segment for the respective individuals. The individual corresponding to the segment on which roulette wheel stopped is selected for the next generation, this process is repeated until the desired number of individuals is selected. It can possibly miss good individuals as there is no guarantee that good individuals will find their way to the next generation (Kumar and Jyotishree, 2012).

Rank selection method, the population is ranked and every individual receives fitness from the ranking. The worst fitness is 1 and the best fitness is N (number of data points). This method is known to have slow convergence rate but keeps up the selection pressure when the fitness variance is very low (Yadav and Sohal, 2017). This method preserves diversity, and hence leads to a successful search.

Tournament selection method selects individuals from the population into a mating pool, where they are used to generate new offspring. An individual with better performance is selected for the next generation.

The selection pressure is the degree to which better individuals are selected: more selected individuals lead to more better individuals for next generation (Deepa and Sivanandam, 2010). It should be noted that high selection pressure can lead to premature convergence, producing a suboptimal solution (Dasha, 2023).

2.4.3 Crossover Operators

Crossover operators generates new offspring by combining the genetic information of two or more parents (Yadav and Sohal, 2017). This research primarily focus on RGAs, hence the crossover operations reviewed in this literature are the ones implemented in RGAs. Common crossover operators for RGAs that are used up to date include the Blend Crossover (BLX), Simulated Binary Crossover (SBX), and Arithmetic Crossover.

Blend Crossover (BLX- α)

Blend crossover was initially introduced by (Eshelman and Schaffer, 1993) with the notion of interval schemata for RGAs. Given two parents, P_1 and P_2 , the maximum between the parents, $P_{max} = \max(P_1, P_2)$, and the minimum $P_{min} = \min(P_1, P_2)$ is identified. From there, the difference between parents is calculated, $I = P_{max} - P_{min}$. Afterwards, two new offspring is randomly (uniformly) selected within the range:

$$\text{off}_1, \text{off}_2 \in (P_{min} - I \cdot \alpha, P_{max} + I \cdot \alpha)$$

It should be noted that this crossover operator provides random generation of a new gene based on the neighborhood of the genes in the parents (Herrera et al., 2002). It should be noted that the choice of $\alpha \in [0, 1]$ is important. This is because for larger α , the range of possible offspring is increased, encouraging more exploration of the search space while a smaller α restricts the offspring's genes to a narrower range around the parents' genes. Therefore, to strike an appropriate equilibrium between exploiting good solutions and exploring new potential solutions within the search space, $\alpha = 0.5$ can be selected. This selection was suggested in (Eshelman and Schaffer, 1993) and implemented in recent researches such as (Pinazo, 2024; Saadaoui et al., 2021).

Simulated Binary Crossover (SBX)

SBX is arguably the popular crossover method in continuous domain, including RGAs (Emmerich et al., 2007; Beume et al., 2007). This method generates the next generation offspring based on a probability distribution. The spread factor, β which is the ratio of distance between offspring and difference between values of parents is controlled by the distribution, where $\beta = \frac{|\text{off}_1 - \text{off}_2|}{|P_{max} - P_{min}|}$. In addition to that, η_c , distribution index (a user-defined control parameter), alters the exploration space of the operator. Specifically, a higher value of η_c indicates a higher probability of generating offspring solutions near the parents (Pan et al., 2021). Then, the offspring generated by SBX is given as follows:

$$\begin{aligned} \text{off}_1 &= 0.5(1 + \beta)P_1 + 0.5(1 - \beta)P_2 \\ \text{off}_2 &= 0.5(1 - \beta)P_1 + 0.5(1 + \beta)P_2 \end{aligned}$$

It should be noted that both values of offspring are equidistant from values of parents. In addition to that, the overall probability of creating two offspring values outside the range of value of parents is the same as creating a pair of offspring within the range of value of parents (Chacón and Segura, 2018).

Arithmetic Crossover

The arithmetic crossover method is a linear combination of two parent individuals (Michalewicz, 2013). Two randomly selected individuals produce two distinct offspring using the following equation:

$$\begin{aligned}\text{off}_1 &= \alpha \cdot P_1 + (1 - \alpha) \cdot P_2 \\ \text{off}_2 &= (1 - \alpha) \cdot P_1 + \alpha \cdot P_2\end{aligned}$$

where α is the weight value, governing the dominant individual in the process of producing new offspring. α takes the value between 0 and 1 (Köksoy and Yalcinoz, 2008).

2.4.4 Mutation Operators

The purpose of the mutation operator is to ensure genetic diversity within the population. This is to ensure that all solutions within the search space do not fall into a local optima (Lambora et al., 2019). Common mutation methods for RGAs include uniform mutation, Gaussian mutation, boundary mutation, and swap mutation. To simplify understanding, the components of each chromosome after mutation are represented as Ω , while the components prior to mutation are denoted as ω .

Uniform Mutation

Uniform mutation is a simple mutation that replaces the existing gene of an individual (Rajakumar, 2013). Therefore, given a parent, $P(i) : i = 0, 1, 2, \dots, N - 1$ with N length of offspring, a mutation operator simply replaces the values of $P(i) \sim U(x_{\min}, x_{\max})$ where $x_{\min}$ and $x_{\max}$ are the limits of the solution space. Uniform mutation is especially helpful in steering away from getting stuck in local optima (Song et al., 2021). The simplicity of this uniform mutation process, in addition to its capacity to keep diversity in the population (Ibili et al., 2022), helps make uniform mutation a viable element of evolutionary optimization processes.

Boundary Mutation

The boundary mutation also replaces an existing gene of an individual. However, it replaces the existing gene with the boundary of the solution space. Provided that $x_{\min}$ and $x_{\max}$ are the boundaries of the solution space, the boundary mutation replaces a specific gene in an individual with either $x_{\min}$ or $x_{\max}$ (Rajakumar, 2013). It is mentioned in (Song et al., 2013) that boundary mutations are good at tackling the issue of avoiding local optima and promoting global optima.

Gaussian Mutation

Gaussian mutation perturbs an individual by adding noise from an error following the Gaussian distribution, $N(\mu, \sigma)$. It is worth mentioning that the Gaussian distribution follows the distribution (Squires, 2001)

$$\varepsilon = G(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

A random Gaussian noise is created using the above formula, ε , with μ being the mean and σ as standard deviation. The formal Gaussian operator can be introduced as follows:

$$\text{mut}_{Gaus}(\omega_{i,j}) = \Omega_{i,j} = \omega_{i,j} + \varepsilon_{i,j}$$

These random changes can be a great tool in order to fine-tune solutions (Bassel et al., 2024). It is also mentioned in (Adegboye and Deniz Ülker, 2023) that lower mean values and lower standard deviations allows the algorithm to find the solution well. Given the nature of the small range of values for $\omega_{i,j}$, it shows that having a large standard deviation can lead to significant changes. This may potentially disrupt good solutions.

Swap Mutation

Swap mutation is an effective mutation operator that exchanges two randomly selected components within an individual (Jiang et al., 2023; Eiben and Smith, 2015). It can be implemented on rows, columns, and elements. It can be stated that mutation on individual elements may introduce small changes for ideal solutions.

$$\Omega_{k,l} = \omega_{i,j} \text{ and } \Omega_{l,j} = \omega_{k,l}$$

It can be seen that the mutation functions swaps a random $\omega_{i,j}$ with $\omega_{k,l}$. It has also shown promise in (Deep and Mebrahtu, 2011; Serpell and Smith, 2010), proving to be among the best mutation methods. It is also mentioned to be easy to implement, but may cause premature convergence (Katoch et al., 2021).

Crossover Rate and Mutation Rate

For the mutation rate p_m , large values will cause difficulty controlling the evolution of the population, while a too small mutation rate will cause the premature population (Zhang and Qu, 2021). Furthermore, with researchers such as (Bernard et al., 2015) recommending low mutation rates, p is selected to be about 0.1 to 1. Research like (Bernard et al., 2015; Obitko, 1998) recommends the use of high crossover rates of about 80 – 95%. It has also shown that research using a higher percentage of crossover produces admirable results (Xue et al., 2021; Owais et al., 2020).

On the other hand, studies suggest that adaptive mutation and crossover rate significantly improve convergence while reducing the chances of being trapped in a local optima (Srinivas and Patnaik, 1994). Various approaches have been proposed including a 'messy' approach (Srinivas and Patnaik, 1994), clustering-based adaptive approach (Zhang et al., 2007), and fuzzy-controlled rates (Zhang et al., 2004). Recent

advances in this area also include work done by (Yang et al., 2016) and (Han and Xiao, 2022). These adaptive rates allow the genetic algorithm to automatically cater for the specific needs of the model, hence improving the model itself.

2.4.5 Existing Studies utilizing GA optimization

GAs have been extensively used to improve the performance of classification models. The earlier studies to this include GA with global weighted k -NN (Kelly Jr and Davis, 1991) and GA with piecewise linear segments used in (Bandyopadhyay et al., 1995). Upon achieving good results, it is then implemented in other machine learning and deep learning algorithms, namely, evolving neural network by (Liu et al., 2004), Decision Trees, Random Forest, and k -NN in (Toma et al., 2020) and Convolutional Neural Network (CNN) in (Sun et al., 2020). This suggests that the implementation of GA to optimize machine learning and deep learning algorithms has been transformative in classification tasks.

In the area of software defect prediction, (Manjula and Florence, 2019) utilized the genetic algorithm for feature optimization and applied the Deep Neural Network (DNN) to detect software defects. A work done by (Ayon, 2019) also implemented GA to select software metrics to be fed into deep learning techniques. On the other hand, weights of Artificial Neural Networks (ANN) are optimized for defect prediction in (Kumar and Sathyanarayana, 2015). (Wahono et al., 2014) also utilized GA to optimize parameters of ANN. These results suggests that the GA algorithm has been explored extensively, including in SDP.

2.5 Summary of Reviewed Methods

This section provides the reviewed methods that were used in the field of software prediction. The findings of the methods implemented can be summarized in the Table 2.1.

Table 2.1: Machine Learning and deep learning techniques visited in Background Review			
Model	Advantage	Disadvantage	Reference
Decision Trees	Flexible with imbalanced datasets	Overfitting when grown to full-depth	(Khoshgoftaar et al., 1998; Chinenye et al., 2023)
k -NN	Simple and does not assume the underlying model	Curse of dimensionality	(Khoshgoftaar et al., 2002; Schuh et al., 2014)
SVM	Efficient with high-dimensional data	Computationally intense	(Kulamala et al., 2021)
Naive Bayes	Simple and computationally efficient	Independence assumption limits effectiveness	(Jiang et al., 2008; Ömer Faruk Arar and Ayan, 2017)
Ensemble Methods	Optimize prediction accuracy	Computationally demanding	(Khalid et al., 2023)
Unsupervised Learning	Effective in preprocessing	Heavily depends on parameter tuning and susceptible to outliers	(Alsaeedi and Khan, 2019; Oyewole and Thopil, 2023)
ANN	Simple model	Require parameter tweaking	(Panda, 2019)
DNN	Generalizable and better accuracy	Overfitting, high computational cost	(Manjula and Florence, 2019; Wu et al., 2018)
CNN	Excels at identifying local pattern	Limited performance on big data sets	(Qiu et al., 2022)
RNN & LSTM	Excel at processing sequential data	Vanishing gradient problem	(Liu et al., 2018; Lu and Salem, 2017; Le and Zuidema, 2016)
GNN	Identifies linkages between software metrics	Computationally intense	(Wu et al., 2020b; Wu et al., 2020a)

2.6 Utilization of GA for k -NN optimization

From section 2.4.5, it has been shown that GA provide reliable optimizing classification algorithms. It has been proven to provide great results in this literature. Therefore, moving further into this research focus, the implementation of GA for k -NN is reviewed.

(Li et al., 2020) implemented an algorithm that selects the important features of a high-dimensional data set, using a GA-optimized weighted k -NN model to search for the optimal weight vector. The method is implemented in 13 real data sets, including 6 high-dimensional microarray data sets. The results of the study suggest that the classification performance is improved due to optimization implemented.

Other research implemented GA-optimized weighted k -NN is a research done in (Su et al., 2008), which presented a feature weighting method. Initially, this model weights 35 features in training for an intrusion detection system, and selecting the best few for classification tasks. An overall 0.9742 accuracy was obtained from this optimization.

In addition to that, (Han et al., 2021) implemented a similar method called the GeneticKNN model to identify the best feature weights. It is compared with several other methods, such as Random Forest, XGBoost, and SVM, but still provided the best results. This suggests superiority among other machine learning algorithms.

In addition to identifying the best global weights, a series of studies conducted by Daniel Mateos-García provided insight on genetic algorithms to optimize feature and neighbor importance for k -NN. This can be seen in (Mateos-García et al., 2012) where the weight matrix is optimized using a genetic algorithm and the output prediction is made using a k -NN model (k -LDEDW). From the results of the study, this method provided great results comparing various data sets. In addition to that, the paper (Mateos-García et al., 2016) utilized GA to adjust the contribution of each one of the k nearest neighbors, providing satisfactory results based on various data sets. Also, a Simultaneous Weighting of Attributes and Neighbors (SWAN) method, which is essentially a combination of the previous work, provided a successful behavior approach (Mateos-García et al., 2019).

2.7 Justification of proposed approach

Based on analysis of the existing literature, it can be deduced that in the field of SDP, implementation of k -LDEDW has not yet been carried out. This suggests an avenue to provide an adaptation of this model to use a model that provides good predictive accuracy whilst providing good understanding of the model. This is due to weighting matrix that provides insights on the importance of each software features of each software module, hence offering a more interpretable model compared to standard machine learning and deep learning algorithms such as k -NN and SVM or deep learning algorithms such as CNN or GNN. Developers can leverage this information to understand the unique feature-to-software relationship.

In addition to that, the utilization of adaptive crossover and mutation rates has yet to be implemented in the k -LDEDW algorithm. This enhances the optimization of the weighting matrix, potentially addressing limitations of static evolutionary parameters. Specifically, by dynamically tailoring the weighting matrix, it can potentially uncover better solutions that static rates might miss. Adaptive rates for crossover and mutation allow the exploration of solution space more effectively in the earlier stages of algorithm for diversity (Damia et al., 2021), while later exploiting good solutions with small changes to fine-tune solutions. Adaptive rates will also provide better handling of premature convergence at a local optima problem, hence providing a good convergence rate for best solution (Han and Xiao, 2022).

The k -LDEDW model can also benefit from using multiple mutation methods. Research done by (Wang and Yu, 2024) and (Zhao et al., 2024) provides good results when implementing more than one mutation operator. This is especially important considering the heterogeneous nature of the problem where each class has different feature weightings. For each class (software) in the k -LDEDW algorithm, there is a set of feature weights that introduces heterogeneity into the optimization process. This is because different classes might require different levels of exploration and refinement to identify their optimal weights. Also, certain classes in the data set might have a more complex relationship between different software features, requiring a different approach of the mutation operator.

Chapter 3

Methodology

3.1 Overview

This chapter seeks to provide the methodological approach to systematically address the challenges of existing SDP models. This comprehensive methodology is designed to ensure validity and reliability of results and to facilitate achieving research objectives specified in Chapter 1. The focus is on leveraging the k -LDEDW method, which optimizes the weighting matrix using a genetic algorithm for k -NN classification. This overview provides an in-depth context for the subsequent detailed discussions on problem identification and proposed approach.

3.2 Research Method

This research employs a research methodology to provide a structured approach to address the objectives in Chapter 1. This methodology consists of several phases: problem identification, analysis of existing methods, development of proposed approach, implementation, evaluation. These phases are designed to ensure a comprehensive evaluation of the proposed method and their practical implementations in software defect prediction. The flow chart in Figure 4.6 provides the outline of the step by step process of this research.

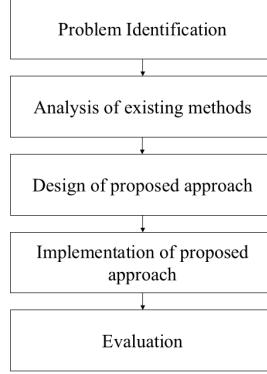


Figure 3.1: Flow chart of research methodology

3.3 Problem Identification

The existing models in this area presents strengths and challenges. This has been discussed in detail in the previous chapter. Despite the advancements of machine learning and deep learning in the field of SDP, k -LDEDW offers several advantages to tackle existing challenges. This can be further defined as follows:

1. **Model Interpretability:** One of the primary issues of existing models is to provide a classification model that produces great predictive power and interpretability of the model. Deep learning models and machine learning models such as k -NN, SVM, CNN, and GNN demonstrate good results in terms of prediction accuracy but do not explain how they software features affect defect predictions. These algorithms mainly function as as black boxes, which makes it difficult for developers to understand how decisions are made. k -LDEDW leverages a weighting matrix that explicitly captures the importance of each feature for each software module, making it inherently more interpretable.
2. **Alignment with Heterogeneous Nature of SDP:** Each software module vary significantly in terms of complexity, feature distributions and defect patterns. In contrast to existing models using global weights, k -LDEDW assigns a unique set of weights for each module, capturing its specific characteristics. This approach makes it suitable for SDP as feature importance varies across different software.
3. **Limitation of Static Optimization in k -LDEDW model:** Despite the model's ability to provide insights on model's decision making and suitability for heterogeneous data sets, the traditional k -LDEDW is limited by its use of static crossover and mutation rates in the GA optimization process. These static rates approach might hinder the optimization process due to constraining exploration and reduce the ability to fine-tune the model at later stages, potentially missing optimal solutions. Adaptive rates dynamically adjusts rates based on state of evolution and address these limitations by improving search efficiency and robustness.

4. **Implementation of single mutation method in k -LDEDW:** In traditional k -LDEDW, a single mutation operator might not accommodate the needs of all individuals. Different software modules might exhibit varying levels of complexity and relationships among features. Relying solely on one mutation strategy limits the algorithm's adaptability to these differences. Introducing multiple mutation methods can introduce greater diversity, allow tailored exploration and enhance robustness in handling the data set with heterogeneous feature-weighting requirements.

3.4 Analysis of Existing Models

As provided in the literature review, the field of SDP has seen significant advancements with the advent of ML and DL techniques. However, an analysis of the current literature identified a number of shortcomings and potential areas for development. This is especially evident when it comes to maintaining high predictive accuracy while allowing for model interpretability. A summary of the results and consequences of the earlier research is provided in this section.

The existing models shows strengths in various aspects. This includes predictive performance. High prediction accuracy is shown by models such as Decision Trees, k -NN, SVM, and DL-based architectures (e.g., CNN, RNN) on a variety of datasets. In addition to that, Ensemble approaches overcome the shortcomings of individual algorithms by combining models to further maximize prediction power. It can also be mentioned that models such as SVM and DNN perform well with high-dimensional data, which is a common characteristic of SDP datasets. Finally, techniques like GNN provide insights into the connections between software projects by using the relational structure of software metrics while unsupervised learning approaches facilitate preprocessing and investigation of latent patterns.

On the other hand, most ML and DL models act as a “black box”, providing little information about how predictions are computed. Because of this, developers find it difficult to comprehend how particular software characteristics affect the likelihood of defects. Although models like CNN and RNN are quite accurate, they are unable to clearly explain how software properties relate to predictions. The k -LDEDW algorithm has yet to be used in SDP. Its weighting matrix offers a unique advantage by capturing important software metrics to predict the state of the software, addressing the gap of interpretability.

Due to SDP datasets being varied, traditional models frequently employ global weights or static parameters across all data points. These methods don’t take into consideration how various software modules or classes differ in the relevance of a feature. By improving global weights, models based on genetic algorithms (GAs), such as k -NN with GA optimizations, have been shown to improve classification performance. These models have been shown to improve classification performance by increasing global weights. However, the ability of these models to effectively explore and exploit the solution space is limited since they often rely on static crossover and mutation rates. In addition to that, the use of a single mutation operator in GA-optimized models limits the algorithm’s capacity to manage a range of feature connections and software complexity. Existing research suggests that adaptive crossover and mutation rates might enhance optimization processes, increasing convergence rates and solution quality. However, k -LDEDW for SDP has not been subjected to these techniques. Some mutation strategies that have been demonstrated to work in a variety of datasets have not yet been investigated in k -LDEDW.

3.5 Design of Proposed approach

This research has been structured in a clear and concise manner, as well as to provide a road map for the research process. The diagram in Figure 3.2 illustrates the sequence of steps followed in this research.

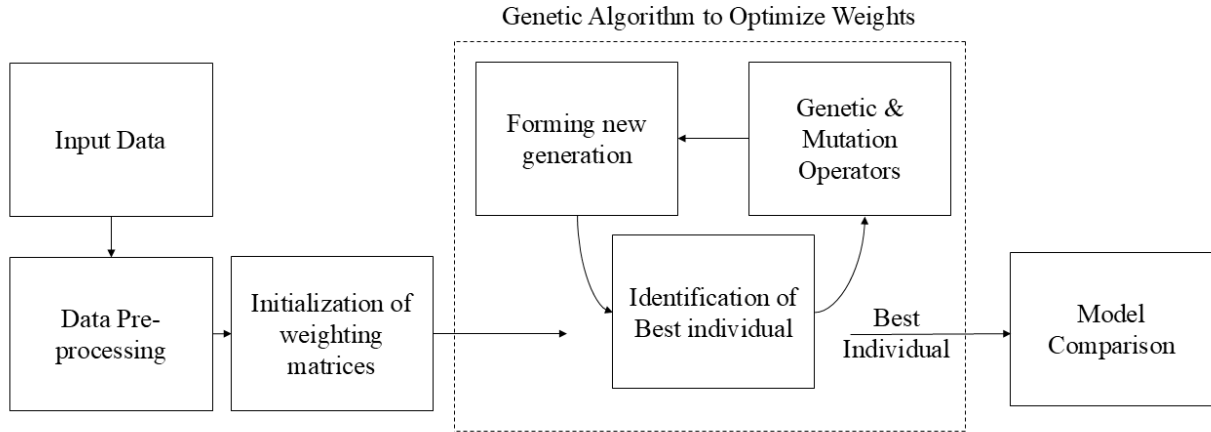


Figure 3.2: Flowchart of k -LDEDW Implementation

3.5.1 Input Data

The first step in this research involves obtaining a data set, which in this case is the ApacheJIT dataset, comprising of buggy and non-buggy commits. It is imported into the Python workspace using Python's Pandas package (pandas development team, 2020). This data set will be further analyzed in the following steps of this algorithm.

3.5.2 Data Pre-processing

Once the data set is loaded into the workspace, it is then pre-processed to enable the algorithm to be implemented optimally. Pre-processing involves cleaning and refining the dataset by removing irrelevant features that do not contribute to the prediction task. In this research data pre-processing includes removing the less relevant features such as Commit ID, project name, and author-date. This step will reduce data noise, enhance computational efficiency and allow the algorithm to focus on more meaningful features.

3.5.3 Initialization of Weighting Matrices

Given the data set with N data points and f features, $D = \mathbb{R}^{N \times f}$, an individual is a weighting matrix of the same dimension, $\dim(\omega) = \dim(D) = (N, f)$. Each weighting matrix entry corresponds as a label to the respective entry of the data set, D . The weighting matrix is initialized with a non-negative real value $\omega_{i,j} \in [0, 1]$ that is less than or equal to 1. Based on equation (2.7), the value 0 in the weighting matrix denotes that the feature of the data point corresponding to it is irrelevant while a value of 1 indicates that it is very important. To provide further clarity, consider a concrete example: a random data set with $N = 3$ and $f = 2$, represented as a 3×2 matrix. An individual (a weighting matrix), ω_i can be associated with this data set and utilized in the classification algorithm. This can be seen below:

$$D = \begin{pmatrix} 1 & 7 \\ 7 & 8 \\ 3 & 0 \end{pmatrix}, \omega_i = \begin{pmatrix} 0.419 & 0.981 \\ 0.940 & 0.560 \\ 0.298 & 0.450 \end{pmatrix}$$

3.5.4 Genetic Algorithm to Optimize Weights

Genetic Algorithms (GA) are a category of evolutionary optimizer inspired by genetics and natural selection (Mathew, 2012). GAs are commonly used to approximate solutions to optimization problems by representing candidate solutions as "chromosomes" which undergo evolutionary processes through genetic operators such as crossover or mutation (Lim et al., 2017). GAs explore the solution space and converge toward optimal or near-optimal solutions for complex problems. For application on matrices in this research, it is aimed to optimize the weighting matrix to provide the best results for the algorithm. To determine an evolutionary algorithm, it is necessary to define an individual encoding, genetic operators, a fitness function and a rule to carry out the generational replacement (Mateos-García et al., 2019). These aspects are defined in the following sections.

3.5.5 Genetic Operators

Genetic operators are mechanisms that are essential to find an optimal solution from the evolution process (AlKhafaji et al., 2020; Albadr et al., 2020). Matrices have specific features that need to be preserved when applying the operators such as structure, dimension and constraints. A constraint imposed on the matrices in this research is that each component of the matrix should remain within $[0, 1]$ as mentioned in 3.5.3. This can be further ensured by implementing a lower bound of 0 and an upper bound of 1 for each weighting matrix after undergoing a genetic operator. Implementing this constraint will allow exploration but at the same time retain stability of the algorithm to prevent unrealistic solutions. It will also provide better intuition when representing probabilities.

The exploration of the solution space is controlled by these genetic operators, which can be classified into crossover and mutation. These genetic operators are designed to explore the solution space while preserving

the structural and dimensional constraints of the matrices. In mutation, small random changes are introduced to the matrix components, while in crossover, two matrices combine their features to generate new solutions. The matrix structure is maintained by ensuring that the offspring matrices have the same dimensions as the parent matrices. The following section provides a better understanding of the genetic operators designed for matrices that is used in this research.

3.5.6 Blend Crossover (BLX- α)

The blend crossover method or BLX- α is a crossover method that is common in the literature (Eshelman and Schaffer, 1993). In the context of this research using matrices, this crossover is applied on each component of parent matrices, generating offspring values by sampling from an extended range defined by the parent values and an expansion factor α . Specifically, given 2 parent matrices ω and ω' , the crossover between these matrices is carried out on their respective i -th row of each matrix. Denoting each elements of each parent as $\omega_{i,j}$ and $\omega'_{i,j}$, the new gene is a real value randomly chosen in the range of $[\omega_{min} - I\alpha, \omega_{max} + I\alpha]$. This is better illustrated in the following figure:

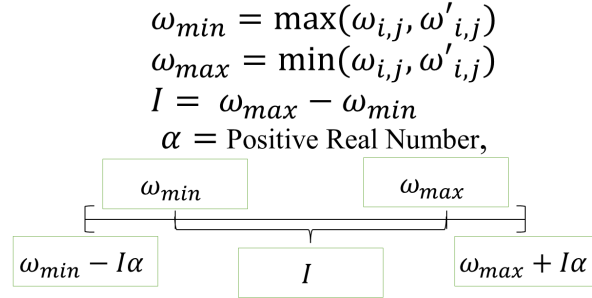


Figure 3.3: BLX- α crossover

As can be seen from figure 3.3, the new offspring is selected randomly within the new range where α controls the gene's variation. This is to make sure that offspring solutions maintain diversity while being closely related to their parents. The crossover operator can be expressed mathematically as $\text{blx-}\alpha(\omega_{i,j}, \omega'_{i,j})$ and follow a uniform distribution of $U(\omega_{min} - I\alpha, \omega_{max} + I\alpha)$ where $\omega_{i,j}$ and $\omega'_{i,j}$ are the two parents. To clarify this operation, consider two 3×2 weighting matrices (ω and ω'), randomly generated using Python using NumPy. Each entry of both matrices undergoes the crossover. For example, the first entries in both matrices participate to generate offspring based on the $\text{blx-}\alpha$ formula. It can be clarified as shown below

$$\omega = \begin{pmatrix} \omega_{1,1} & \omega_{1,2} \\ \omega_{2,1} & \omega_{2,2} \\ \omega_{3,1} & \omega_{3,2} \end{pmatrix}, \omega' = \begin{pmatrix} \omega'_{1,1} & \omega'_{1,2} \\ \omega'_{2,1} & \omega'_{2,2} \\ \omega'_{3,1} & \omega'_{3,2} \end{pmatrix} \Rightarrow \omega_{new} = \begin{pmatrix} \text{blx-}\alpha(\omega_{1,1}, \omega'_{1,1}) & \text{blx-}\alpha(\omega_{1,2}, \omega'_{1,2}) \\ \text{blx-}\alpha(\omega_{2,1}, \omega'_{2,1}) & \text{blx-}\alpha(\omega_{2,2}, \omega'_{2,2}) \\ \text{blx-}\alpha(\omega_{3,1}, \omega'_{3,1}) & \text{blx-}\alpha(\omega_{3,2}, \omega'_{3,2}) \end{pmatrix}$$

From this, it can be seen that each of the components of ω_{new} is obtained from implementing the $\text{blx-}\alpha$

formula on the first entry of ω and ω' . This creates a set of two new weighting matrices (offspring) that can be used for the next iteration.

It should be noted that the choice of α is important. This is because for larger α , the range of possible offspring is increased, encouraging more exploration of the search space while a smaller α restricts the offspring's genes to a narrower range around the parents' genes. Therefore, to strike an appropriate equilibrium between exploiting good solutions and exploring new potential solutions within the search space, $\alpha = 0.5$ can be selected. This selection was suggested in (Eshelman and Schaffer, 1993) and implemented in recent research such as (Pinazo, 2024; Saadaoui et al., 2021).

3.5.7 Mutation

Mutation operators modify one or more elements of a matrix to explore new areas of the search space and prevent premature convergence to suboptimal solutions (Jalali Varnamkhasti and Vali, 2023). By randomly altering parts of a matrix, mutation ensures the algorithm does not become overly reliant on existing genetic material, thereby maintaining genetic diversity and avoiding stagnation in local optima (Jalali Varnamkhasti and Vali, 2023). The design of mutation operators for matrices should balance exploration and exploitation (Salgotra and Mirjalili, 2024), introducing changes that are impactful enough to explore new possibilities but in the same time maintain the feasibility of solution. Mutation methods such as uniform mutation, Gaussian mutation and swap mutation have been selected for this research, and tailored to effectively make changes to the matrix while preserving structural integrity. A detailed description of mutation operator applied in this research is described in 4.3

3.5.8 Adaptive Mutation and Crossover Rates

Adaptive mutation and crossover rates are essential components of the genetic algorithm used in this research, enabling dynamic adjustments to enhance exploration and exploitation during the evolutionary process. These strategies are inspired by the work of (Han and Xiao, 2022). Further description of the implementation of mutation and crossover rates can be found in Section 4.2 where the mathematical explanation, pseudocode, and an example is provided.

3.5.9 Integration of Adaptive Rates

The adaptive mutation and crossover rates work in tandem to guide the evolutionary process. At each generation, the fitness of all individuals is evaluated, and the rates are computed for each individual. These rates are then used to determine the likelihood of applying mutation and crossover operators, balancing the need for diversity and convergence. By incorporating these adaptive strategies, the genetic algorithm achieves improved performance and robustness in optimizing the weighting matrices.

3.5.10 Fitness function

The fitness function plays a crucial role in genetic algorithms, as it provides a quantitative measure of the quality of each individual in the population — guiding the evolutionary process toward optimal or near-optimal solutions (Eiben and Smith, 2015). The goodness of a chromosome in solving the problem is measured by the fitness function. The purpose of a fitness function is to seek the global optimal solution (Han and Xiao, 2022). In this research, the fitness function used is the Matthew’s Correlation Coefficient (MCC) evaluation metric. Although there are many widely used evaluation metrics that can be used to measure performance such as F_1 -score, recall and accuracy (Hosseini et al., 2017; Malhotra, 2015), it has been pointed out by statisticians that these metrics are problematic (Hernández-Orallo et al., 2012; Powers, 2020).

The Matthews Correlation Coefficient (MCC) is a robust evaluation metric that is used widely for its balanced assessment of classification performance, even in imbalanced data sets (Boughorbel et al., 2017; Chicco and Jurman, 2020). An imbalanced data set means that one class significantly outnumbers the others. In section 3.6.1, it will be made clear that the JIT data set is imbalanced. MCC is employed as the fitness function to guide the genetic algorithm toward solutions that optimize the overall quality of classification. This section explains the motivation, mathematical formulation, and practical implementation of MCC as the fitness function.

Justification of Fitness Function

As was mentioned, imbalanced datasets pose significant challenges for traditional metrics like accuracy, which can provide misleading results by overemphasizing the majority class. MCC addresses this issue by taking all four elements of the evaluating properties - positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) — into a single metric. It reflects the strength and direction of the relationship between predicted and actual labels, making it ideal for evaluating classifiers on imbalanced datasets.

MCC Metric

The choice of accuracy as the fitness metric aligns with the primary objective of this research: to optimize the weighting matrix such that the algorithm maximizes classification performance. It is defined mathematically as (Skorikov and Momen, 2020):

$$MCC = \frac{(TP \cdot TN) - (FP \cdot FN)}{\sqrt{(TP + TN)(TP + FN)(TN + FP)(TN + FN)}} \quad (3.1)$$

The values of MCC can range from -1 to 1 . A value of 1 indicates that the classification model provides perfect prediction. Value of 0 indicates that the model performs no better than random prediction while -1 suggests that the model provides completely incorrect predictions.

It should be noted that the practical considerations of implementing MCC as the fitness function is multipurpose. This is because the MCC can be calculated using Python’s *sklearn.metrics.matthew_corrcoef*,

which eases the computational process. Also, the interpretation of MCC values is helpful to the algorithm, as negative values can be used as feedback to iteratively improve.

Characteristics of the Fitness Function

In this research, MCC is used as the fitness function to assess the candidate solutions (the weighting matrices) during the evolutionary process of the genetic algorithm. For each iteration, the data set used for prediction, along with the candidate weighting matrix, is passed through the k -NN classifier and evaluated as fitness level in terms of the MCC value of such predictions. The fitness level is also recorded in terms of MCC – the higher the values, the better - and this advances the perfecting of the algorithm towards solutions that are optimal.

Formally, letting ω represent the weighting matrix used in the classification and D as the data set, denote predicted output as $Pred(\omega, D)$ and actual output as $True(D)$. The fitness function $F(\omega, D)$ can be represented as follows:

$$F(\omega, D) = MCC(Pred(\omega, D), True(D)) \quad (3.2)$$

By employing the MCC as the fitness function, this research ensures not just accurate solutions but also balanced and meaningful classification outcomes. This is particularly useful when working with imbalanced datasets.

3.5.11 Generational Replacement

Generational replacement simply means replacement of individuals from one generation to the next. In this research, the elitist strategy is chosen in which the single best individual is chosen to move to the next generation without being affected by crossover or mutation operator (Yusoff and Othman, 2018). Here, the elitist strategy is particularly useful as it ensures the best solutions are preserved while still allowing variation through crossover and mutation, preventing premature convergence and allowing further optimization. The elitist method has shown to help achieve better convergence in genetic algorithm (Deb et al., 2002). After the best individual is chosen, denoting N as total of individuals, the population of the next generation are formed as follows:

- the following $C - 1$ of the best individuals are cloned from the previous generation
- The remaining $N - C$ individuals are created using crossover operator.
- All individuals except the first best performing individual is subjected to mutation operator with probability of p .

It should be noted that to select the individuals from previous generation for crossover operation, (Mateos-García et al., 2019) mentioned that using this algorithm, there are no significant differences when using either tournament or roulette-wheel method.

3.5.12 Iteration Process of Genetic Algorithm

For the purpose of clarification, the following Figure 3.4 has been provided:

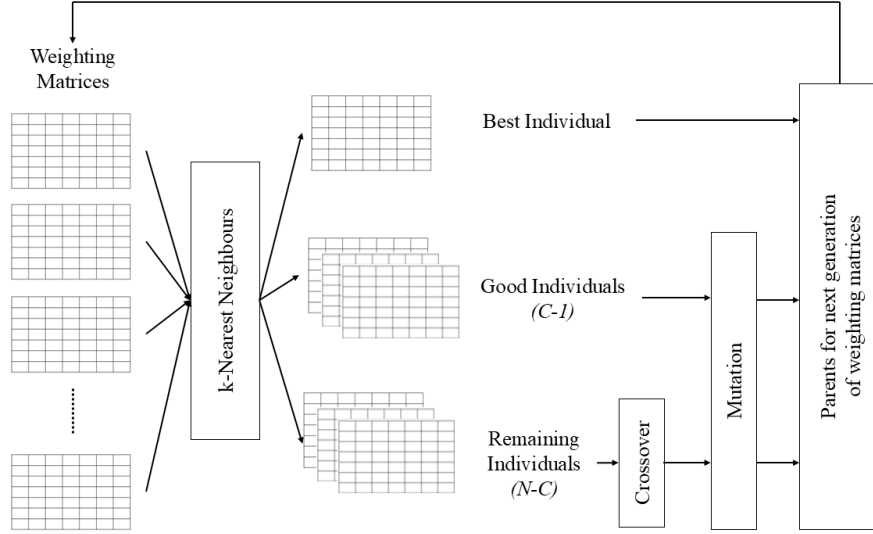


Figure 3.4: Iteration process of genetic algorithm adapted from (Mateos-García et al., 2012)

The iteration phase consists of three fundamental tasks: Feeding the dataset with the individual weighting matrix to the k -NN model, identification of the best individuals, and applying the genetic algorithm operators to introduce changes to generation. These key steps allows the k -LDEDW algorithm to explore the feature space to find the optimal solution hence, providing the best classification results.

Feed to k -NN Classification Model

From the previous generation (except for the first generation), each individual's fitness is evaluated by feeding through the k -NN classification model. This process involves using the individuals as weighting matrices applied to the dataset. This enables the k -NN model to classify data points based on weighted distances from equation 2.7. The robust MCC evaluation metric ensures each individual's predictive performance is accurately measured, enabling a fair comparison based on the quality of classification achieved.

Identify Best Individuals

After the individuals have been evaluated, the genetic algorithm selects the fittest individual based on the highest MCC score. The selected individual provided the best performance in terms of accuracy and predictive reliability. This individual will be selected for the next generation, ensuring superior genetic material is preserved. This process improves the population's overall performance in future iterations. The following $C - 1$ best individuals are selected through tournament selection method as was discussed in Section 3.5.11.

Apply Crossover and Mutation Operator

This step is to form new parents from the offspring of the previous generation. The fittest individual will be selected for the next generation. The best few selected for cloning, $C - 1$, and the remaining individuals, $N - C$, created using the blend crossover operator. It should be noted that all individuals other than the fittest individual are subjected to the mutation operator.

3.6 Implementation

This section provides details on the software environment and hardware used to implement the k -LDEDW algorithm. It describes the tools, libraries, and computational resources that were utilized in this research.

In terms of software environment, the Python version used for this implementation is version 3.11.11. This is chosen due to the available libraries to support machine learning, data processing, and numerical computations. The key libraries and packages used in this research include NumPy (Harris et al., 2020) for numerical operations, pandas for data set manipulation, Scikit-learn (Wes McKinney, 2010) for evaluation metrics such as MCC, Random package to generate random selection and probabilities, and Collections (Counter) for k -NN classifier. For a more efficient management of Python environments, Anaconda Navigator version 24.11.1 is used. Jupyter Notebook version 7.2.2 is used as the primary development environment to implement the algorithm. This is chosen due to its interactivity which simplifies iterative coding and visualization.

The hardware environment used to implement this algorithm uses two laptops. The first one is a personal laptop, Ideapad 3 15IAU7 with 12th Gen Intel(R) Core wand 16.0 GB RAM. This laptop consists of 512 GB SSD storage. The second one is provided by Oxford Brookes University is DELL Latitude 5520, with 11th Gen Intel(R) Core and 16.0 GB RAM, consisting of 256 GB SSD storage. This is because sufficient memory is critical for handling the large ApacheJIT data set and the intermediate matrices.

3.6.1 Data

This research utilizes the ApacheJIT data set which is a comprehensive resource for defect prediction (Keshavarz and Nagappan, 2022). The data set contains data from software changes across various popular Apache projects, distinguishing between non-buggy and bug-inducing commits. The data set is comprised of a total of 106,674 commits, of which 28,239 are bug-inducing commits and 78,435 clean commits. The magnitude of the data set makes it very well suited for machine learning applications, which rely on large training sets to generalize historical patterns and predict future outcomes. The features of the data set is given (Haldar and Capretz, 2024a) as shown in Table 3.1.

In order to balance computational feasibility with relevant experimentation, in this research, 10% of the ApacheJIT data set is used. The limitations of the available hardware resources, specifically memory (RAM) and processing capacity (CPU), led to the choice to employ a reduced portion of the dataset. It

Table 3.1: Feature of ApacheJIT Dataset

Features	Description
commit_id	Commit ID
Project	Project name
Author_date	Author date
Fix	Whether or not the change is defect fix
LA	Number of lines added
LD	Number of lines deleted
ND	Number of directories touched
NF	Number of files touched
NS	Number of subsystems touched
Entropy	Distribution of modified code across each file
NDEV	Number of distinct developers touched files
Age	The average time from the last change
NUC	Number of unique changes in files
EXP	Number of change author experience
REXP	Developers' recent experience
SEXP	Developers' subsystem experience
Buggy	Selected commit is buggy or not

would take a lot more time and computer resources to run the entire dataset using the k -LDEDW method, which uses matrix operations and iterative genetic optimization. This method preserves the reliability and generalizability of the results while allowing the experiments to be finished in a reasonable amount of time by randomly selecting 10% of the dataset. This guarantees that the chosen subset stays typical of the original data distribution.

To make model building and assessment easier, the chosen data is separated into subsets for training and testing. The k -LDEDW method is optimized using the training data, which makes up 80% of the sampled dataset. This entails training the k -NN classifier and creating the weighting matrix using the adaptive genetic algorithm. To assess the model's performance, the remaining 20% of the collected dataset is set aside as testing data. This testing subset is subjected to the optimized k -LDEDW model, and evaluation metrics like Matthews are calculated by comparing the model's predictions with the actual labels. The full implementation, including data preprocessing, model training, and evaluation scripts, is available at the following repository: <https://github.com/imran-red/k-Label-Dependant-Evolutionary-Distance-Weighting-kLDEDW-for-Software-Defect-Prediction/tree/main/kLDEDW>.

This research ensures that the findings remain significant in spite of resource constraints by carefully choosing and segmenting the data set. By using improved computer capabilities, research in the future

might apply this methodology to the whole data set with better computational resources.

3.6.2 Random Sampling Method

In this research, a random sampling strategy was adopted to create training and testing splits from the ApacheJIT dataset, rather than partitioning the data based on project boundaries which is commonly used in Within-Project Defect Prediction (WPDP) and Cross-Project Defect Prediction (CPDP). This decision was rooted in several key reasons within the structure of the dataset, and the goal of this study.

this is mainly due to the complication of commits within the ApacheJIT dataset. It is a collection of commits from multiple Apache projects with Just-in-Time bug labels. However, many of these projects are sparse in terms of defect-inducing commits and uneven in size and feature coverage.

WPDP is typically more reliable for large, self-contained datasets where full history of one project is available. ApacheJIT contains fragments of many projects with insufficient temporal depth per project. it would be severely limited to the amount of data available for both training and testing. Hence, it is insufficient for a robust model learning or evaluation.

CPDP on the other hand involve training on one or more projects and testing on a completely different one. This reflects on model generalizability. but assumes a degree of homogeneity between projects. This will lead to distribution mismatch which leads to a weaker model validity. This can be seen in prior studies such as (Vouillon and Di Cosmo, 2013) and (Kondo et al., 2020) where CPDP often suffers from poor performance unless domain adaptation or transfer learning are used.

These limitations motivates the use of random sampling which offers a pragmatic compromise. It ensure sufficient training data by pooling all commits while retaining label imbalance which allows evaluation of performance on both clean and buggy commits. It also allows the model to learn feature-weight interactions across varying instances which aligns with the goal of k -LDEDW. A temporal constraint is also imposed which partially addresses the WPDP philosophy to ensure that training data always precedes testing data. This means that older data are used in training to predict new and future data.

3.7 Model Evaluation

This section provides the evaluation metrics used to compare existing models that have been applied to the ApacheJIT data set, and also an overview of the Shapley values for model interpretability.

3.7.1 Evaluation Metrics

The comparison between the machine learning models used on this dataset can be quantified using evaluation metrics. The evaluation metrics used to compare models in this research are Accuracy, Precision, Recall, f_1 -score, AUC and MCC. The calculation of each metric can be given as follows (Skorikov and Momen, 2020):

$$Accuracy = \frac{TP + TN}{FP + TP + FN + TN} \quad (3.3)$$

$$Precision = \frac{TP}{FP + TP} \quad (3.4)$$

$$Recall = \frac{TP}{TP + FN} \quad (3.5)$$

$$f_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \quad (3.6)$$

Recall that MCC is used as the fitness function of the genetic algorithm in equation 3.1. The AUC evaluation metric is the area under the receiving operating characteristic (ROC) curve. The ROC curve is the graph of true positive rate (TPR) and false positive rate (FPR) over different thresholds. the AUC metric varies from 0 to 1, with 1 being the perfect classifier. A value of 0.5 suggests that the classifier performs no better than random guessing, while lower than 0.5 suggests that the classifier is likely wrongly predicting the outputs (Keshavarz and Rodríguez-Pérez, 2024)

Accuracy can be described as the ratio of correctly predicted instances to total instances, indicating the overall correctness of the model. Equation 3.4, Precision, evaluates the proportion of true positive predictions among all positive predictions, showing how reliable positive classifications are while Recall evaluates the proportion of true positive predictions among actual positive instances, reflecting the model's sensitivity. Finally, the f_1 -score calculates the harmonic mean of Precision and Recall, balancing both metrics.

3.7.2 Shapley Values

Shapley values, which are rooted in cooperative game theory, have become a strong technique to interpret machine learning models. They provide a fair allocation of contributions from individual features toward a model's prediction, to ensure transparency and interpretability (Molnar, 2020; Lundberg, 2017). This method is gaining traction in modern machine learning applications based on its ability to provide meaningful insights on how different features influence model predictions (Heuillet et al., 2022). It is essential to understand that Shapley values are attributed to each feature in the data set. For this research, the ApacheJIT data set, the features analyzed are features used in the implementation after data pre-processing.

3.7.3 Understanding Shapley values

Shapley Values in Linear Models

Starting with the simple linear model, decomposing each feature’s contribution is straightforward. Consider a linear model defined as

$$\hat{f}(x) = \beta_0 + \sum_{j=1}^p \beta_j x_j \quad (3.7)$$

where x_j are the feature values and β_j are their corresponding coefficients representing the influence of each feature on the prediction. The next step is to identify the contribution of each features. The contribution ϕ_j of feature j to the prediction $\hat{f}(x)$ is determined by the following expression:

$$\phi_j(\hat{f}) = \beta_j x_j - \mathbb{E}(\beta_j X_j) \quad (3.8)$$

where $\mathbb{E}(\beta_j X_j)$ represents the average contribution of feature j across the dataset. Summing all feature contributions leads to the following equation:

$$\sum_{j=1}^p \phi_j(\hat{f}) = \hat{f}(x) - \mathbb{E}(\hat{f}(X)) \quad (3.9)$$

General Shapley Value Definition

The general Shapley value for feature j is computed using the following mathematical formulation:

$$\phi_j(val) = \sum_{S \subseteq N \setminus \{j\}} \frac{|S|!(p - |S| - 1)!}{p!} [val(S \cup \{j\}) - val(S)] \quad (3.10)$$

where S represents a subset of features, $val(S)$ is the model output when only features in S are used, and p is the total number of features in the dataset. A brief explanation of this equation can be understood by considering the $\frac{|S|!(p - |S| - 1)!}{p!}$ as the “weight” accounting for subsets of features. The factor $[val(S \cup \{j\}) - val(S)]$ can be thought of as the “value” factor which provides the value of contribution of model’s performance with respect to subset of the features.

Properties of Shapley Values

Shapley values exhibit unique mathematical properties that justify their use as a fair attribution method. One of the most useful characteristics of this method is efficiency. This is because it ensures the sum of feature contributions equals the difference between model’s prediction for a specific instance, and the average prediction across the data set (Abuzinadah et al., 2023). For instance, if a model predicts 80 and the average prediction is 60, the extra 20 points will be fully explained by the contributions of the features. This property guarantees all contributions are accounted for and nothing left unexplained.

Other than that, symmetry is also an essential property of Shapley values. This means that if two features contribute equally to the model’s prediction in all scenarios, they should be assigned equal Shapley values (Casajus, 2019). This prevents bias in attribution when features have equivalent impact to predictions.

The next property of the Shapley value is the dummy property. The dummy property ensures that features with no influence on the model’s prediction receive a Shapley value of zero. This means that if the model essentially ignores a feature - as it plays no role in prediction task - its contribution to the prediction is always 0, regardless of the subset it is in (Sundararajan and Najmi, 2020).

Finally, the additivity property ensures that when combining multiple models, the Shapley values of individual features from each model can be summed to obtain the final Shapley value for the combined model (Yang et al., 2024). This property basically means that if you have multiple models, you can combine their Shapley values by adding them together to explain the combined model.

3.7.4 Model Comparison

The individual demonstrating the fittest performance in the final generation is selected as the best individual. This optimal weighting matrix is then applied to the customized k -NN model with distance weighting (Equation 2.7) along with the data set to identify its performance with other evaluation metrics. Thus completing the k -LDEDW algorithm.

Based on these evaluation metrics, the enhanced k -LDEDW model will be extensively compared with the k -NN model which acts as a baseline model. The k -LDEDW model will also be compared with other models that have been implemented to the dataset namely JITGNN, JITLine, Clustering techniques and JIT techniques. This will be further discussed in the following chapter.

3.7.5 Temporal Cross-Validation for Just-In-Time Defect Prediction

In machine learning experiments, n -fold cross-validation is commonly used to evaluate model performance by splitting the dataset randomly into n partitions to ensure each subset is used for testing at least once. This traditional approach may lead to temporal leakage in terms of using ApacheJIT dataset, where software commits are time-coded. This is because evaluating a model on data that comes chronologically before its training data violates the realistic deployment scenarios (Kondo et al., 2020).

This is addressed by implementing temporal cross-validation method that respects the ordering of commits based on their chronology in the ApacheJIT dataset. Specifically, the commits within the dataset are sorted by timestamp to provide a guideline. Then, a rolling origin evaluation was applied where the model was trained on commits prior to a certain timestamp and evaluated on the next block of commits. This process is iterated over multiple windows within the dataset to evaluate generalization across software history. This ensures the test set contains commit that are truly after the training set. This step is taken to consider the real world application in software defect prediction.

Chapter 4

Implementation and Results

4.1 Overview

The adaptive mutation rate, which enables mutation probabilities of individuals to be dynamically adjusted based on their fitness, is a significant advancement offered in this work. Traditional genetic algorithms often rely on fixed mutation rates, which can lead to suboptimal performance because they either overexplore the search space or prematurely converge to local optima. The adaptive mutation rate overcomes these limitations and preserves the equilibrium between exploration and exploitation by modifying the mutation probability for each individual in the population.

The multiple mutation approach is another enhancement introduced in this work to further provide an effective and versatile genetic algorithm. Instead of relying on a single mutation technique that might not be suitable for all individuals or search space circumstances, the multiple mutation method randomly selects one of many mutation procedures for each person. The balance between exploration of feature space with focused exploitation, along with adaptive rates, ensures a robust search process that avoids premature convergence while refining high-quality solutions.

4.2 Adaptive Mutation and Crossover Rates

Adaptive mutation and crossover rates are essential components of the genetic algorithm used in this research, enabling dynamic adjustments to enhance exploration and exploitation during the evolutionary process. These strategies are inspired by the work of (Han and Xiao, 2022).

4.2.1 Adaptive Crossover Rate

The adaptive crossover rate adjusts the probability of crossover dynamically based on the fitness of the parents. This ensures that high-performing individuals preserve their genetic material while low- and moderately-performing individuals contribute to exploring new combinations. The crossover probability is defined by (Han and Xiao, 2022) as:

$$p_c = \begin{cases} K_1 \frac{f_{max} - f_c}{f_{max} - f_{min}} & f_c \neq f_{max}, f_{min} \\ K_2, & f_c = f_{min} \\ K_3, & f_c = f_{max} \end{cases} \quad (4.1)$$

where p_c is the crossover probability, f_c corresponds to the parent with better fitness, f_{min} and f_{max} is the minimum and maximum fitness of the entire population, and K_1, K_2 , and K_3 are constants between 1 and 0. K_1 is set to be 0.9, as was recommended by studies such as (Bernard et al., 2015) to provide diversity. K_2 is set to be 0.5 to allow low fitness individual to escape local optima and K_3 is set to 0.1 to reduce crossover rate among performing individuals. This dynamic adjustment ensures that individuals with better fitness are less likely to undergo crossover, preserving their high-quality traits, while poorly performing individuals are more likely to participate in crossover to generate new offspring.

4.2.2 Adaptive Mutation Rate

The adaptive mutation rate is designed to dynamically adjust the probability of mutation based on the fitness of an individual relative to the population. This mechanism ensures that low- and moderately-performing individuals are subjected to more significant mutations to explore new solutions, while higher-performing individuals are altered less frequently to preserve advantageous traits. The mutation probability is defined by (Han et al., 2021) as:

$$p_m = \begin{cases} K_4 \frac{f_{max} - f_c}{f_{max} - f_{min}} & f_c \neq f_{max}, f_{min} \\ K_5, & f_c = f_{min} \\ K_6, & f_c = f_{max} \end{cases} \quad (4.2)$$

Where K_4, K_5 , and K_6 are constants between 1 and 0. K_4 is set to be 0.1, as was recommended by studies such as (Bernard et al., 2015) to provide diversity. K_5 is set to be 0.05 to allow low fitness individual

to escape local optima and K_6 is set to 0.001 to reduce crossover rate among performing individuals. This adaptive strategy promotes diversity and prevents premature convergence by encouraging exploration among individuals that are not performing well. The following pseudocode in Figure 4.1 for adaptive mutation rates allows a simple understanding of how the method is constructed, along with its purpose.

```

Input:
- acc: Array of fitness scores for all individuals in the
population
- K4: Maximum mutation rate factor
- K5: Minimum mutation rate for the least fit individual
- K6: Minimum mutation rate for the most fit individual
Output:
- adaptive_pm: Array of mutation rates for all individuals
Steps:
1. Compute the maximum fitness value:
   f_max = max(acc)
1. Compute the minimum fitness value:
   f_min = min(acc)
1. Initialize an empty list to store mutation rates:
   adaptive_pm = []
4. For each fitness score f_c in acc:
   a. If f_c == f_max (the highest fitness score):
      - Set the mutation rate pm = K6 (minimum mutation rate
      for best-fit individual)
   b. Else if f_c == f_min (the lowest fitness score):
      - Set the mutation rate pm = K5 (higher mutation rate
      for least-fit individual)
   c. Otherwise (for intermediate fitness scores):
      - Compute the mutation rate pm as:
      pm = K4 * (f_max - f_c) / (f_max - f_min)
      - This adjusts pm proportionally based on how close f_c
      is to f_min or f_max.
   d. Append pm to adaptive_pm.
5. Convert adaptive_pm to a NumPy array.
6. Return adaptive_pm as the output.

```

Figure 4.1: Adaptive Mutation Rates Pseudocode adapted from (Han and Xiao, 2022)

It should be noted that a similar implementation of this code can be used for adaptive crossover rates, as they are similar with different values for K_i .

4.2.3 Example of Adaptive Rates

To provide further clarification on implementing adaptive rates, an example is provided. For example, assuming the fittest individual in the solution space is $f_{\max} = 0.85$, and the low performing individual of $f_{\min} = 0.3$. For an arbitrary individual with fitness $f_c = 0.83$, the mutation probability can be given as

$$p_m = \frac{f_{\max} - f_c}{f_{\max} - f_{\min}}$$

$$p_m = \frac{0.85-0.83}{0.85-0.3} \approx 0.0036$$

This suggests that good-performing individuals, are less prone to being affected by crossover or mutation operator. Hence, this ensures that mutation and crossover rates align better with the relative performance of individuals in the population

4.3 Multiple Mutation Implementation

It was mentioned that selected mutation methods for this research are uniform mutation, Gaussian mutation and row swap mutation. This is because uniform mutation allows the diversification of solutions by making broad explorations. Gaussian mutation on the other hand is an ideal mutation operator for fine-tuning, as it introduces small changes to matrix entries. Finally, row swap mutation balances diversity by changing rows within a matrix while preserving structure. To simplify understanding, the components of each chromosome after mutation are represented as Ω , while the components prior to mutation are denoted as ω .

4.3.1 Uniform Mutation

Uniform mutation is a simple genetic operator that applies changes with a constant probability for each gene in any individual. This enables the population to gain diversity. It declares that for every element $\omega_{i,j}$, this mutation replaces its value with the random number from the uniform $U(0, 1)$ distribution with probability p . This means that for a probability p ,

$$\text{mut}_{\text{unif}}(\omega_{i,j}) = \Omega_{i,j} \sim U(0, 1) \quad (4.3)$$

By doing so, they can ensure a good distribution of genetic diversity across individuals, which assists with solution space exploration. Uniform mutation is especially helpful in steering away from getting stuck in local optima (Song et al., 2021). The simplicity of this uniform mutation process, in addition to its capacity to keep diversity in the population (Ibili et al., 2022), helps make uniform mutation a viable element of evolutionary optimization processes.

4.3.2 Gaussian Mutation

As was discussed in section 2.4.4, where Gaussian mutation adds noise from an error following the Gaussian distribution, $N(\mu, \sigma)$. It worth mentioning that the Gaussian distribution follows the distribution (Squires, 2001)

$$G(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \quad (4.4)$$

In the context of this research, a random Gaussian noise is created using the NumPy and random packages in python with mean, $\mu = 0$ and standard deviation $\sigma = 0.5$. The choice of σ is to ensure that the extreme randomness introduced by this mutation operator is reduced. Recall that in Section 3.5.5, the numerical

constraint will be imposed after each mutation. After obtaining the error function, ε , we can formally introduce the Gaussian mutation operator as the following

$$\text{mut}_{Gaus}(\omega_{i,j}) = \Omega_{i,j} = \omega_{i,j} + \varepsilon_{i,j} \quad (4.5)$$

After obtaining the mutated weighting matrix, it has to undergo post-processing to ensure the numerical constraint is imposed. This is to ensure each entry remains within the $\omega \in [0, 1]$. One of the clear advantages of this mutation operator is that it introduces small and localized changes. These random changes can be a great tool in order to fine-tune solutions (Bassel et al., 2024). It is also mentioned in (Adegboye and Deniz Ülker, 2023) that lower mean values and lower standard deviations allows the algorithm to find the solution well. Given the nature of the small range of values for $\omega_{i,j}$, it shows that having a large standard deviation can lead to significant changes. This may potentially disrupt good solutions.

4.3.3 Swap Mutation

Section 2.4.4 provides a brief overview of swap mutation. The swap mutation swap components within the mutated matrix. It can be implemented on rows, columns, and elements. It can be stated that mutation on individual elements may introduce small changes for ideal solutions. However, mutation at a row level introduces controlled variability and reduced disruptions, which is useful when the matrix represents relationships or weights that are interdependent within rows and columns. Which applies to this particular research, the dataset will be visited in section 3.6.1. Therefore, it is justified to utilize row swap mutation, which at a probability p , randomly selects two rows within a weighting matrix and swaps them. This can be further described with a weighting matrix $\omega_{i,j}$ with f features:

$$\text{mut}_{swap}(\omega_{i,j}) = \text{mut}_{swap} \begin{pmatrix} & & \dots & & \\ \omega_{k,1} & \omega_{k,2} & \omega_{k,3} & \dots & \omega_{k,f} \\ & & \dots & & \\ \omega_{l,1} & \omega_{l,2} & \omega_{l,3} & \dots & \omega_{l,f} \\ & & \dots & & \end{pmatrix} = \begin{pmatrix} & & \dots & & \\ \omega_{l,1} & \omega_{l,2} & \omega_{l,3} & \dots & \omega_{l,f} \\ & & \dots & & \\ \omega_{k,1} & \omega_{k,2} & \omega_{k,3} & \dots & \omega_{k,f} \\ & & \dots & & \end{pmatrix} \quad (4.6)$$

Which can be further simplified to be for all $j \in f$, the operator simply:

$$\Omega_{k,j} = \omega_{l,j} \text{ and } \Omega_{l,j} = \omega_{k,j} \quad (4.7)$$

It can be seen that the mutation function swaps a random row k in the weighting matrix $\omega_{i,j}$ with row l . The implementation in this research utilizes Python's NumPy library. Two pairs of indices, k and l are chosen uniformly at random from the matrix dimensions. The rows at these indices are then swapped. One of the benefits of this method is it adheres to the numerical constraints hence no post-processing is required. Also, implementing this mutation operator preserves the structure of the matrix.

For the purpose of providing insights with implementation of multiple mutation operators, the pseudocode from Figure 4.2, inspired by (Wang and Yu, 2024), is provided. While our implementation differs, it highlights an alternative approach to the problem. This shows how different types of mutation methods are applied to different individuals.

```

Input:
- res_acc: Array of fitness scores for all individuals
- total_weights: Array of weighting matrices (population)
- Mutation Methods: "adaptive", "swap", "uniform"

Output:
- Updated total_weights after applying mutations

Steps:
1. Compute adaptive mutation rates:
   adaptive_pm = adaptive_mutation_rate(res_acc)
2. Iterate over each weighting matrix in total_weights:
   For i = 0 to len(total_weights) - 1:
     a. Retrieve the mutation probability for the current individual:
        pm = adaptive_pm[i]
     b. Generate a random number r in [0, 1]:
        If r <= pm (apply mutation with probability pm):
          i. Randomly select a mutation method:
             mutation_type = Random choice from ["adaptive", "swap", "uniform"]
          ii. Apply the selected mutation:
             If mutation_type == "adaptive":
               - Add Gaussian noise to the weighting matrix:
                 total_weights[i] += Random values from N(0, 0.)
             Else If mutation_type == "swap":
               - Perform a row or column swap within the weighting matrix:
                 total_weights[i] = swap_mutation(total_weights[i])
             Else If mutation_type == "uniform":
               - Apply uniform mutation to the weighting matrix:
                 total_weights[i] = uniform_mutation(total_weights[i])
3. Return the updated total_weights.

```

Figure 4.2: Implementation of Multiple Mutation Methods Pseudocode

4.4 Results of Adaptive k -LDEDW Implementation

This section presents the performance evaluation of the k -LDEDW and k -NN models applied to the ApacheJIT data set. Both models are assessed using the evaluation metrics described in Section 3.7.1, providing a comprehensive comparison of their ability to predict correctly and effectiveness in software defect prediction tasks.

4.4.1 k -LDEDW Model Performance

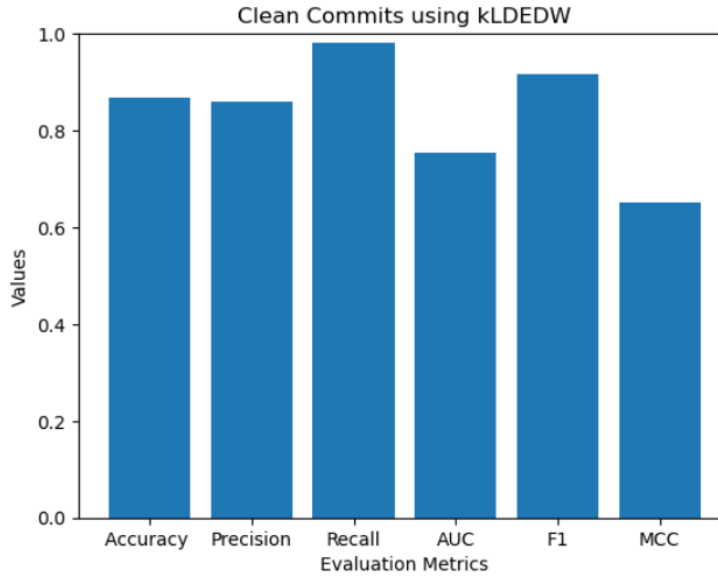


Figure 4.3: Performance of k -LDEDW method with clean commits

Upon implementing the k -LDEDW algorithm on the ApacheJIT dataset, the performance of this model on clean commits (not bug-inducing commits) can be shown in Figure 4.3. The accuracy of the k -LDEDW model is 0.87, indicating the model’s ability to identify clean commits accurately. The precision for clean commits is 0.86, which implies that most of the predicted clean commits were accurately classified. Next, the recall for clean commits is 0.98, highlighting the model’s outstanding ability to detect clean commits when they are present. This results in a high f_1 -score of 0.92, reflecting a balanced performance between precision and recall. The Matthews Correlation Coefficient (MCC) of 0.65 and an AUC of 0.75 further confirms the model’s robust performance in distinguishing clean commits.

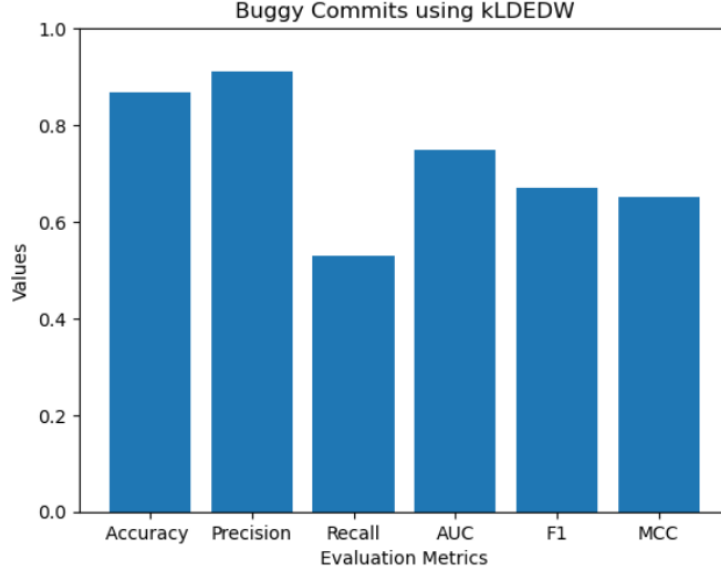


Figure 4.4: Performance of *k*-LDEDW method with buggy commits

On the other hand, the model’s performance for buggy commits (bug-inducing commits) is described by the bar chart in Figure 4.4. The accuracy of the *k*-LDEDW model for buggy commits is 0.87, indicating a good classification performance. The precision for buggy commits is 0.91, showing that the majority of predicted buggy commits were identified correctly. However, the recall for buggy commits is 0.53, suggesting a poor ratio of TP to $TP + FN$ as shown in Section 3.7.1. The model also yields an f_1 -score of 0.67, reflecting a reasonable balance despite the recall limitation. The Matthews Correlation Coefficient (MCC) of 0.65 and an AUC of 0.75 further shows the model’s effectiveness in identifying buggy commits, although there is room for improvement in recall performance.

4.4.2 Performance Analysis Across Different k -values

To investigate the effect of the k parameter in this research, the k -LDEDW model is evaluated across multiple k values, $k = 1, 3, 5$ and 7 . This is because from the review in Chapter 2, it mentions that k plays a critical role in balancing bias and variance. Therefore, a comparative analysis was conducted on the ApacheJIT dataset using the same training and testing splits. The performance of each value is measure using the Matthews Correlation Coefficient (MCC). The following figure illustrates the cumulative MCC score over the different values of k mentioned:

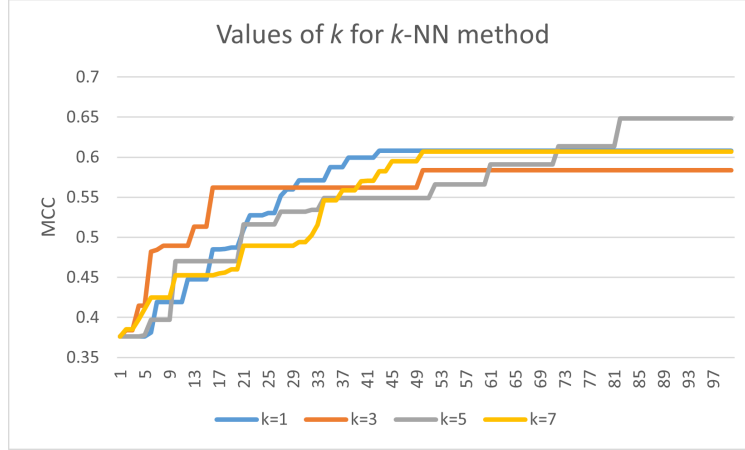


Figure 4.5: MCC of each k value throughout iterations

From the figure, the following observations can be extracted:

- $k = 1$ (blue line): This value showed the most consistent rise in MCC value, showing stability at 0.608. However, it also exhibited the static behavior in the later iterations which suggests overfitting.
- $k = 3$ (orange line): This value demonstrated moderate performance. While it shows the steepest rise at the early stages, it began plateauing at 0.584. It also showed after several improvements on early iterations, the model demonstrated limited improvement in the later stages.
- $k = 5$ (gray line): This value of k provided the highest finale MCC value of 0.65. Although the convergence is slightly slower compared to smaller k values, the performance was more stable and consistent across iterations even at the later stages. This suggests better generalization and reduced sensitivity to noise.
- $k = 7$ (yellow line): This value of k performed less competitively, stabilizing at 0.606, it demonstrated limited gains over time

From the results discussed above, it can be indicated that $k = 5$ provides the most effective balance between model complexity and generalization ability. This is because with smaller k values, they exhibited rapid convergence to relatively high MCC early in the training process, but also showed fluctuations and

instability in later iterations. This can be seen by the ability for these values to reach an optima at an early stage, and begin to plateau afterwards. This behavior is an indication that the model might be overfitted on the training set, causing it to be less effective in generalizations.

In contrast, $k = 5$ shows a smooth graph throughout the series of iterations. This is done by aggregating the influence of multiple neighbors which reduces the likelihood that single misleading commit will dominate the decision. This in turn leads to a more stable and reliable prediction across the dataset. Although the convergence is relatively slower compared to lower k values, it can be deduced that the final performance was consistently better.

$k = 7$ demonstrated weaker performance, with convergence at 0.606, and limited improvements at later stages. This performance is an indicator of underfitting where the model is too generalized and fails to capture the patterns in the data.

As a result, $k = 5$ was selected for the final model implementation, based on its performance and stability. This selection aligns with the model’s design principle to assign adaptive, instance-level weights, and reinforces its suitability to handle complex software defect data.

4.4.3 Evaluation under Temporal Cross-Validation

As was discussed in Section 3.7.5, implementing the traditional cross-validation method will introduce temporal leakage within the ApacheJIT dataset. To address this, a temporal cross-validation method is performed to better suit the ApacheJIT defect prediction scenarios.

After sorting the dataset chronologically by commit timestamp, a sliding window approach is used where approximately 10% of the large ApacheJIT dataset were used as training and testing set. This approach ensures that the model is always trained on past data, and future commits are never used in training. In summary, the evaluation method mimics actual deployment where new commits are continuously obtained with time.

Based on the previous section of selecting $k = 5$, the k -LDEDW model achieved an average MCC of approximately ≈ 0.58 across all temporal folds. Although it is lower than the 0.65 MCC in the original evaluation, this is expected and it highlights and demonstrates the challenge of generalizing across evolving software. It is also notable that the model exhibited low variance in MCC across folds, suggesting that it provides a stable generalization. This result further confirms the effectiveness of this model, even under temporal constraints.

4.4.4 Convergence Analysis of Genetic Algorithm and Weight Stability

An essential aspect to validate the model is to ensure that the Genetic Algorithm used to optimize feature weights demonstrates proper convergence behavior and provides a set of stable weights over time. This section presents the analysis of the convergence process and the consistency of the final weights in supporting the reliability of optimization.

The convergence of the GA was evaluated by tracking the MCC at each generation. Each generation within the GA consisted of a population of weight matrices. The best performing weight matrix from each generation is recorded to observe the algorithm's progression towards an optimal solution. This evaluation is demonstrated in the following figure:

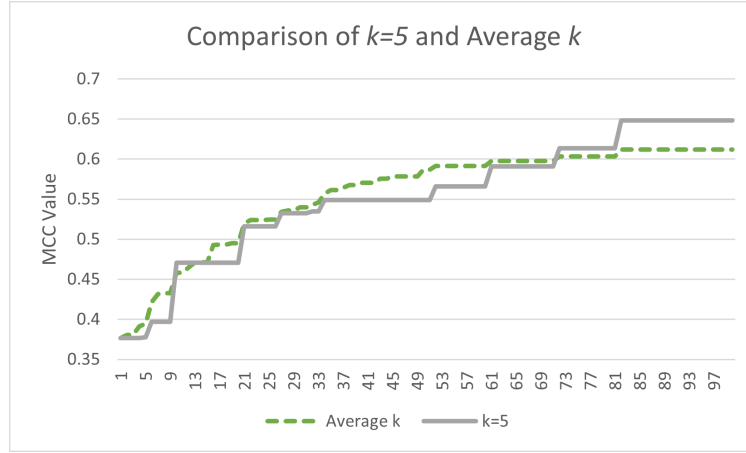


Figure 4.6: Comparison of $k = 5$ with the average MCC value of other k values

Figure above shows the progression of MCC values throughout each iteration for:

- $k = 5$ (solid gray line)
- average predictions across multiple k values (dashed green line)

Both values show a steady improvement over time, with early generations showing a steep gains as the GA explore the solution space. Over time, it can be see that the improvement slows which indicate convergence. It should be worth noting that although in the early stages there are higher improvement rate, but it happens gradually, suggesting reducing chances of overfitting.

The fixed $k = 5$ approach consistently higher final MCC while displaying step-like gains which reflects bursts of significant optimization . On the other hand, the average k approach stabilizes earlier, reaching the final MCC with smoother progression and fewer jump. This suggests that although it converges faster, it may not converge as effectively in reaching the global optimum. This demonstrates that while averaging across k values may reduce volatility and variance in the short term, the fixed $k = 5$ model allows for more targeted optimization and ultimately achieves better predictive performance.

In addition to monitoring via MCC convergence, the stability of learned feature weights was examined. Throughout the early generations, the weight matrices exhibited high variability which is driven by wide search space. In the later stages, the weights produced by top-performing individuals became increasingly similar which is consistent with the goal of implementing GA. This suggests that the evolutionary process has reached a local or global optimum. It follows that the decisions made by the model are not arbitrary but rather grounded in stable patterns in the data.

The convergence analysis demonstrates that the genetic algorithm applied in k -LDEDW effectively navigates the search space, leading to performance improvements over time. The comparison in Figure 4.X shows that the fixed $k = 5$ setting outperforms the averaged k strategy in terms of final MCC while retaining convergence stability. Additionally, the stabilization of feature weights reinforces the interpretability and robustness of the method, making k -LDEDW not only well performing but also suitable for deployment in practical software defect prediction.

Chapter 5

Evaluation & Discussion

In this chapter, the results of the proposed k -Label Dependent Evolutionary Distance Weighting (k -LDEDW) on the ApacheJIT dataset will be evaluated. The performance of the model is compared against the k -NN model which acts as a baseline model. Comparison with other models including Clustering techniques and JIT bug prediction techniques (Jadhav et al., 2023) and also JITGNN and JITLine from (Keshavarz and Rodríguez-Pérez, 2024) follows. Finally, this section provides a comparison of contribution of each feature in making predictions using Shapley values.

5.1 Model Comparison

In this section, the comparison among the models will be thoroughly analyzed. This section is divided into two: comparison with GNN models, and comparison with machine learning models. This is due to the results of (Keshavarz and Rodríguez-Pérez, 2024) that differentiate the evaluation for clean and buggy commits, while the work by (Jadhav et al., 2023) is evaluated using overall data.

5.1.1 Comparison with GNN models

Table 5.1: Model Comparison for Clean Commits

Model/Evaluation Metric	MCC	Accuracy	Precision	Recall	f_1 -score	AUC
k -LDEDW	0.65	0.87	0.86	0.98	0.92	0.75
k -NN	0.17	0.72	0.42	0.28	0.33	0.66
JITGNN	0.38	-	0.93	0.69	0.79	0.81
JITLine	0.36	-	0.93	0.65	0.77	0.81

From Table 5.1, it can be seen how different models perform based on evaluation metrics for clean commits. The k -LDEDW model has an accuracy of 0.87, indicating a strong classification ability. Its precision is 0.86

and recall is 0.98 which shows its effectiveness at identifying clean commits and balancing false positives and false negatives. The k -LDEDW model’s f_1 -score is 0.92 which indicates that it balances recall and precision. As for precision metric, JITGNN and JITLine models perform better with 0.93 precision. This means the Graph Neural Network based models are good at predicting clean commits out of all actual predictions. The AUC score of 0.75 for k -LDEDW shows a good level of distinction between clean and buggy commits, but both JITGNN and JITLine models have an AUC score of 0.81 suggesting that they perform better for this evaluation metric. The MCC is 0.65 which is much higher than other classifiers which means there is good positive correlation between predicted and actual labels.

Table 5.2: Model Comparison for Buggy Commits

Model/Evaluation Metric	Accuracy	Precision	Recall	f_1 -score
k -LDEDW	0.87	0.91	0.53	0.67
k -NN	0.72	0.78	0.87	0.82
JIT GNN	-	0.37	0.78	0.50
JITLine	-	0.35	0.80	0.49

From Table 5.2, it can be seen that the evaluation metrics for different models to identify buggy commits. The k -LDEDW model has the highest accuracy (0.87) and precision (0.91) among all models, meaning it minimizes false positives and correctly identifies buggy commits out of all predicted buggy commits. This means that k -LDEDW exhibits good performance at providing reliable and good predictions. Nevertheless, k -LDEDW missed a large number of real defective commits, as evidenced by its recall of 0.53. The f_1 -score, which weighs precision and recall, is impacted by the poor recall score and comes out at 0.67. Although k -LDEDW has a high precision, it should be improved to detect more bug commits due to its recall trade-off.

However, with the maximum recall of 0.87, k -NN is the most effective model for identifying the majority of real buggy commits. Its f_1 -score of 0.82 is the highest of all the models, indicating that it outperforms k -LDEDW in terms of precision and recall balance. However, despite having a high recall, k -NN may generate more false positives as its accuracy (0.72) and precision (0.78) are lower than those of k -LDEDW.

The performance of the deep learning models JIT GNN and JITLine is comparatively worse. Both models exhibit low f_1 -scores (0.50 and 0.49, respectively) and low precision (0.37 and 0.35, respectively). Despite having respectably high recall scores (0.78 and 0.80, respectively), their poor precision indicates that they frequently make inaccurate predictions about defective commits.

In conclusion, k -LDEDW is good for predicting high-confidence bug commits since it has the best accuracy and precision. Despite its reduced precision, k -NN is effective at detecting the majority of buggy commits. Although the deep learning models show promise in recall, they still need to be further refined for accuracy and overall effectiveness.

5.1.2 Clustering techniques and JIT bug prediction techniques

Table 5.3: Model Comparison with Clustering Techniques and JIT bug prediction techniques

Model/Evaluation Metric	Accuracy	Precision	Recall
k -LDEDW	0.87	0.88	0.76
k -NN	0.72	0.6	0.58
Clustering techniques	0.93	-	0.68
JIT bug prediction techniques	0.81	0.50	-

Table 5.3 summarizes the results and evaluation metrics which gives an understanding of different models. The overall performance of the k -LDEDW model is rather good as illustrated in Table 4 with an accuracy rate of (0.87). Although this is somewhat below the clustering methods (0.93), k -LDEDW still has strong classification capabilities. The precision of k -LDEDW is also the highest with 0.88, which demonstrates that it generates fewer false positives and can identify clean commits reliably. Thus, given a setting where predictions corresponding to clean commits are being made, the results suggests that k -LDEDW is better at accurately identifying clean commits. Moreover, it has a recall of 0.76 (the highest among other models), which means the model has the ability to identify most actual clean commits.

In comparison, the k -NN and JIT bug prediction techniques perform the worst in all evaluation metrics. The k -NN model’s accuracy is 0.72, precision is 0.6 and recall value is 0.58. This indicates a weaker overall classification performance. Other than that, JIT bug prediction techniques shows good accuracy of 0.81, but with a very poor precision value of 0.50.

In conclusion, k -LDEDW maintains a high-level of classification accuracy with great balance between precision and recall over all commits identified. This makes it a competitive model for classifying commits. Nonetheless, the most accurate solution is provided by clustering techniques and extensions of this work could be utilized to exploit both approaches as they have been shown to complement each other.

5.2 Model Explainability

In this section, models k -LDEDW and k -NN are compared using the Shapley values for each feature. The Shapley values provide an insight on the importance of each feature to the model’s prediction, hence allowing a deeper understanding of interdependencies between features, for each classification model.

5.2.1 Shapley values for k -LDEDW

From Figure 5.1, the highest Shapley feature value can be obtained from LA (Lines added) with the normalized value of 1. This suggests that this feature is highly influential in making predictions, hence suggesting that the amount of new code added significantly impact the predictions. Other features that also strongly

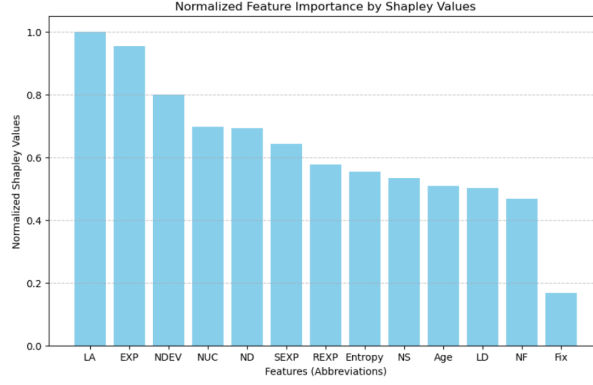


Figure 5.1: Shapley values for feature in k -LDEDW

contribute include EXP (Number of change author experience) and NDEV (Number of developers) indicating that these features are critical in capturing the nature of the data set.

Other than that, moderately contributing features such as ND (Number of directories touched), SEXP (Developer's subsystem experience) and REXP (Developer's recent experience) have considerable relevance for classifications tasks. This reflects the relevance of developer interaction patterns and recent experiences.

Other features such as LD (Lines deleted) and Age (Average time from last change) did not play a big role compared to the top features. This indicates the they might have limited predictive power within the k -LDEDW method. Finally, the lowest Shapley feature value is Fix (Whether or not the change is a defect fix), which shows that this feature has very small contribution to prediction.

5.2.2 Shapley values for k -NN

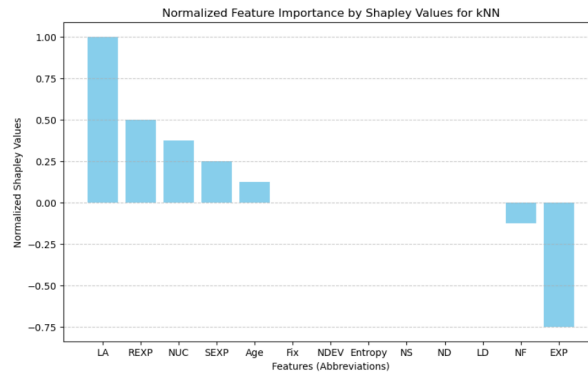


Figure 5.2: Shapley values for features in k -NN

As can be seen in Figure 5.2, LA (Lines Added) has the highest normalized Shapley value, indicating it is the most important feature influencing k -NN predictions. This is very similar compared to Shapley values in k -LDEDW. Followed by moderately contributing features such as REXP (Developer's recent experience) and

NUC (Number of unique changes in files) which suggests that recent activity and diversity in contributions are influential in this model. SEXP (Developer’s subsystem experience) and Age (The average time from the last change) also contribute positively but are influential critical than LA, REXP, and NUC.

Also, the k -NN algorithm has features with negative Shapley values. It should be said that negative Shapley values play a big role in model interpretability. This is because, in the pursuit of finding clean commits, a negative value decreases the possibility of a commit being clean. Hence, this suggests that higher values of negatively valued features are associated with a higher likelihood of predicting buggy commits. This is shown by EXP (Number of change author experience) and NF (Number of files touched). The high negative value of EXP features suggests that higher values for this feature reduces the likelihood of predicting a data point as clean for k -NN model.

Finally, features like Entropy, NS (Number of subsystems touched), ND (Number of directories touched), LD (Number of lines deleted), and Fix (Whether or not the change is a defect fix) have Shapleys value of 0. This indicates that these features are of no importance in predicting an output. This further shows that kNN does not rely on these features for its predictions in this context.

5.2.3 Comparison of Shapley Values

In terms of significant features, both models agree that LA is an important aspect for the classification task. This indicates that the size of code changes plays a important part in predicting the output of a commit. For k -LDEDW, this is followed by EXP, which shows that the code author’s experience is highly valued by this model. However, for k -NN, it is followed by REXP and NUC. This shows that even though both models value the importance of LA, k -LDEDW relies on overall experience of the developer more than the recent experience and narrow aspects of activity.

Other than that, an aspect that is worth commenting on is the feature distribution. From k -LDEDW’s Shapley values, it is clear that feature importance is more evenly distributed across a larger set of features. This suggests that k -LDEDW captures a more diverse aspect of the data set, potentially making it more robust. On the other hand, the k -NN model places significant importance for few features namely LA, REXP, NUC and EXP. Although these features contributed to the prediction phase, having no importance to features such as Fix, LD, NS, ND, NDEV and Entropy reflects that this model ignores some potentially relevant features. It can be deduced that k -LDEDW provides a more comprehensive evaluation of features, while k -NN prioritizes a smaller subset of critical features.

The even distribution of Shapley values shows that k -LDEDW is sensitive to multiple aspects of the data set, reflecting the purpose of its dynamic weighted distance-based approach. k -NN’s sharp reliance of few features on the other hand indicates its susceptibility to localized data patterns. This can lead to the model being more vulnerable to bias in data sets with incomplete information or imbalanced data sets. Overall, it can be seen that k -LDEDW offers a more holistic evaluation of the data set as compared to k -NN.

The next observation that can be made concerns the negative Shapley values. For the k -LDEDW imple-

mentation, all features give positive Shapley values. This suggests that every feature contributes positively to the predictions. However, for the k -NN model, features like EXP and NF shows negative values, which indicates that if the values of these features increase in the dataset, the likelihood of certain outcomes is reduced. It should be noted that the presence of a negative contribution in k -NN suggests it penalizes specific features in making predictions while k -LDEDW integrates all features as positively contributing factors.

In terms of interpretability, k -LDEDW provides a more balanced insight into features contributions. This may make it preferable as it eases the process of understanding how different features collectively impact predictions while k -NN focuses mainly on dominant features, which makes it simpler to interpret but also narrows the scope of information considered from the data set. The more inclusive feature utilization of the k -LDEDW algorithm suggests that it might generalize better to data sets with different feature distributions. This is because it handles data efficiently across various features. On the other hand, k -NN's reliance on a few features may limit its adaptability across diverse data sets.

5.3 Significance of Results in Software Defect Prediction

The results presented in this research shows the effectiveness of the proposed k -LDEDW model in improving software defect prediction, particularly in Just-In-Time (JIT) setting using the ApacheJIT dataset. The following paragraphs explain several perspectives that prove the significance of these findings.

The k -LDEDW model outperformed other existing models across multiple evaluation metrics such as MCC, recall, precision and f_1 -score. This indicates that the proposed method is more effective in distinguishing between buggy and clean commits, especially in imbalanced datasets where traditional models often struggle. In the context of SDP, higher prediction accuracy enables earlier detection of bugs, hence reducing costly debugging later in the development stage. By identifying these defect-prone commits, developers can prioritize testing and code review efforts more effectively and efficiently.

k -LDEDW method also provide personalized feature importance. Unlike models that apply global weights or rely on static feature importance, this model generates instance-specific weights optimized by genetic algorithm. This further leads to a more nuanced understanding of what features contribute to defects in specific contexts. Such granularity is especially useful in large-scale systems where different parts of the code database may follow different development patterns. In practice, this helps software developers, project managers, and quality assurance engineers to gain deeper insights into why certain commits may be more likely to cause defects, moving beyond black-box predictions.

In terms of adaptability and robustness, the experiments with multiple k values and the implementation of temporal cross-validation demonstrates the model's adaptability. The model performs well not only in random splits but also under realistic time ordered validation, reinforcing its robustness to deployed in live development environments.

Chapter 6

Threats to Validity

6.1 Internal Validity

Internal validity threats refer to uncontrolled factors that might influence our experimental results. One main threat is the data pre-processing steps. We ensured that the step to split data and encoding is consistent across both models, but potential biases from these steps cannot be fully eliminated.

Other than that, another internal validity threat is related to model training and evaluation. Each model is run three times, and the results are averaged to reduce variance caused by random initialization. However, differences in library implementations and computing environments could still introduce minor inconsistencies.

Finally, the performance metric implementation is another possible source of threat. Well-established libraries, including *scikit-learn* were used for metric calculation. Hence, the evaluation process is tested multiple times to make sure the results were accurate and can be reproduced.

6.2 External Validity

External validity threats involve the extent to which the results of this research generalize beyond the specific data sets and experimental setup. We used Apache the JIT data set which is a cross-project JIT dataset with historical change data from 14 popular Apache projects (Keshavarz and Nagappan, 2022). While this improves comparability, results may not generalize to other domains or datasets with different features. This is due to the software metric provided in this data set which might not be similar to the ones used in other studies.

Furthermore, the comparison models we chose may have specific assumptions that do not generalize to other models. This is because the compared models used in this research are deep learning models (JITGNN and JITLine) and other machine learning algorithm (clustering techniques and JIT bug prediction

techniques). The underlying assumptions in these models may not generalize to other models. Similarly, for model interpretation between k -LDEDW and k -NN, the feature importance evaluation depends solely on the Shapley values framework, which may behave differently for other models beyond k -LDEDW and k -NN.

6.3 Construct Validity

Construct validity threats concern the degree to which the experiments conducted in this research truly measure what they intend to measure. In this research, we compare the k -LDEDW algorithm with k -NN and other existing models using performance metrics such as accuracy, precision, recall, f_1 -score, AUC, and MCC. We also investigate how both k -LDEDW and k -NN models interpret feature importance using Shapley values.

Regarding the dataset used, recent research (Rodríguez-Pérez et al., 2020; Rodriguez-Perez et al., 2020) showed a group of bugs caused by external factors such as name changes in requirements or APIs. These bugs are grouped as extrinsic bugs. From the SZZ algorithm (Śliwerski et al., 2005) that was used to construct this data set, it cannot detect the bugs and causes. Therefore, the results of this research cannot be applied in the context of extrinsic bugs.

Chapter 7

Conclusion & Future Work

7.1 Conclusion

This research has explored the development and evaluation of the k -LDEDW algorithm, which integrates a dynamic evolutionary weighting system with k -NN for software defect prediction. The proposed method was implemented on the ApacheJIT dataset and extensively compared against k -NN, Graph Neural Network (GNN) models (JITGNN and JITLine), clustering techniques, and JIT bug prediction techniques.

In terms of performance evaluation, The k -LDEDW model demonstrated strong classification capabilities, achieving high accuracy and precision across both clean and buggy commit predictions. Specifically, k -LDEDW consistently outperformed k -NN in most evaluation metrics for clean commits, balancing precision of 0.86 and recall 0.92 with an f_1 -score of 0.91. Similarly, k -LDEDW excelled in precision (0.91) for buggy commits, even though its recall value of 0.53 indicated some limitations in detecting all buggy commits. These results reflect k -LDEDW's ability to provide high-confidence predictions, particularly for clean commits, while highlighting areas for potential improvement in recall. In comparison, k -NN displayed notable strength in recall for buggy commits (0.87), making it suitable for identifying a larger proportion of buggy commits, though at the cost of reduced precision and accuracy. Deep learning-based GNN models (JITGNN and JITLine) demonstrated promising recall values but struggled with low precision and f_1 -scores, suggesting that they require further optimization for more consistent predictions. Clustering techniques achieved the highest accuracy among the compared models, but limited metrics were reported, restricting a deeper comparative analysis.

For model explainability this research investigated the feature importance of k -LDEDW and k -NN, providing insights into their interpretability using Shapley values. Both models identified Lines Added (LA) as the most influential feature, indicating the importance of code additions in predicting outcomes. However, k -LDEDW exhibited a more balanced distribution of feature importance across multiple features, such as Developer's Experience (EXP), Number of Developers (NDEV), and Developer's Subsystem Experience

(SEXP), reflecting its robust utilization of diverse data characteristics. Conversely, k -NN focused more narrowly on dominant features like LA, Recent Experience (REXP), and Number of Unique Changes (NUC) while disregarding others, such as Fix and Lines Deleted (LD), which may reduce its robustness and adaptability to varied datasets. Additionally, k -NN's negative Shapley values for some features highlighted its tendency to penalize certain attributes, offering an a different point of view as compared to k -LDEDW's uniformly positive contributions.

Finally the generalizability and scalability of k -LDEDW is studied. The more inclusive feature utilization of k -LDEDW, along with its dynamic weighting mechanism, suggests that it generalizes well across datasets with diverse feature distributions. This adaptability may make it a valuable approach for broader software defect prediction tasks. In contrast, k -NN's reliance on a few dominant features indicates its susceptibility to localized data patterns, potentially limiting its performance in data sets with imbalanced or incomplete information.

7.2 Mapping Research Questions to Thesis Sections

To provide clarity on how each research question (RQ) was addressed, the following table maps the RQs introduced in Section 1.3 to the corresponding sections where they are investigated and answered.

Table 7.1: Mapping of Research Questions to Thesis Sections

Research Question	Thesis Section(s)	Explanation
RQ1.a : To what extent does the k-LDEDW method perform on the ApacheJIT dataset?	Sections 4.4.1, 5.1	Presents and interprets model performance using MCC, precision, recall, and f_1 -score.
RQ1.b : How well does the k-LDEDW method distinguish between clean and buggy commits?	Section 4.4.1	Provides per-class metric breakdown, evaluating prediction quality for both buggy and clean commits.
RQ2 : How does k-LDEDW compare to existing software defect prediction models?	Sections 5.1.1, 5.1.2	Empirical comparisons with GNN-based models, clustering techniques, and traditional JIT approaches.
RQ3 : To what extent do Shapley values provide interpretability of the model’s predictions?	Sections 5.2.1–5.2.3	Uses Shapley values to explain feature contributions and compares interpretability with baseline models.

This mapping confirms that all research questions were addressed through a structured experimental pipeline and substantiated by both quantitative results and interpretability analysis. It should be noted that while RQ1.a and RQ1.b may appear similar, they address distinct but complementary aspects of model evaluation. RQ1.a focuses on the overall performance of the k-LDEDW method across the entire dataset, using global metrics such as MCC and f_1 -score to assess the general effectiveness of the model. In contrast, RQ1.b specifically examines the model’s ability to differentiate between buggy and clean commits, which is particularly important in imbalanced datasets like ApacheJIT. A model may achieve high overall performance while still failing to correctly identify buggy commits which is a recognized limitation in software defect prediction. Therefore, addressing both questions ensures a comprehensive evaluation that captures not only predictive strength but also the reliability of the model in identifying the most critical instances.

7.3 Contribution

The main contribution of this research is to implement adaptive genetic algorithm with k -NN classifier in SDP that addresses the knowledge gaps in terms of model interpretability and predictive accuracy. Specifically, this research contributes through the following ways.

This research pioneered the implementation of k -LDEDW in the field of SDP. By utilizing its distinct weighting matrix, this work suggested that the enhanced k -LDEDW model offers both high predicted accuracy and a thorough comprehension of feature-to-software correlations. k -LDEDW provides a detailed and comprehensible perspective of the defect prediction process by explicitly assigning and interpreting feature importance for every software module, in contrast to compared models such as GNN, clustering techniques and JIT bug prediction techniques.

Other than that, adaptive crossover and mutation rates are incorporated into the genetic algorithm (GA) part of k -LDEDW. This dynamic modification improves the weighting matrix's optimization by preserving variety in the early phases by permitting more extensive investigation. This enhancement also concentrates on fine-grained exploitation to improve solutions in subsequent phases. Thus, resolving the issue of early convergence and raising the pace of convergence overall.

Furthermore, multiple mutation operators are added to the GA to further improve k -LDEDW's resilience and flexibility. The varied character of SDP datasets is addressed by this innovation, since various software modules frequently call for differing degrees of investigation and improvement. This addition also enhances the algorithm's capacity to manage intricate inter-feature connections by utilizing a variety of mutation techniques suited to various optimization requirements.

Finally, in terms of the interpretability of k -LDEDW in SDP, this research combines adaptive GA algorithms with the dynamic weighting matrix of k -LDEDW. Better comprehension and confidence in the model's predictions are made possible by the actionable insights the resultant model gives developers on the significance of particular traits for every software module.

7.4 Future Work

This research used dynamic evolutionary weighting and feature evaluation to show the k -LDEDW algorithm's promise for software defect prediction. Even though the outcomes were encouraging, there are a number of areas that can be improved.

This research implements a genetic algorithm focusing on a single objective: the MCC value. Multi-objective optimization frameworks can be used to simultaneously optimize two or more objectives in order to achieve optimal solution. Utilization of evolutionary algorithms also have produced good results in solving various optimization problems (Hua et al., 2021; Tian et al., 2021).

In addition to that, the implementation of the SWAN algorithm in (Mateos-García et al., 2019) suggests that an adjustment of contribution of neighbors plays a role in classification of k -NN algorithm. It is

mentioned that the synergy of the two sets of weights help improve the classification accuracy. This direction provides the utilization of adaptive crossover and mutation rates, and multiple mutation operators to study the improvements can be made in the model.

References

- Abu Alfeilat, H. A., Hassanat, A. B., Lasassmeh, O., Tarawneh, A. S., Alhasanat, M. B., Eyal Salman, H. S., and Prasath, V. S. (2019). Effects of distance measure choice on k-nearest neighbor classifier performance: a review. *Big data*, 7(4):221–248.
- Abuzinadah, N., Kumar Posa, S., Alarfaj, A. A., Alabdulqader, E. A., Umer, M., Kim, T.-H., Alsubai, S., and Ashraf, I. (2023). Improved prediction of ovarian cancer using ensemble classifier and shaply explainable ai. *Cancers*, 15(24):5793.
- Adegboye, O. R. and Deniz Ülker, E. (2023). Gaussian mutation specular reflection learning with local escaping operator based artificial electric field algorithm and its engineering application. *Applied Sciences*, 13(7):4157.
- Aguzzi, J., Costa, C., Fujiwara, Y., Iwase, R., Ramirez-Llorda, E., and Menesatti, P. (2009). A novel morphometry-based protocol of automated video-image analysis for species recognition and activity rhythms monitoring in deep-sea fauna. *Sensors*, 9(11):8438–8455.
- Albadr, M. A., Tiun, S., Ayob, M., and Al-Dhief, F. (2020). Genetic algorithm based on natural selection theory for optimization problems. *Symmetry*, 12(11):1758.
- Albattah, W. and Alzahrani, M. (2024). Software defect prediction based on machine learning and deep learning techniques: An empirical approach. *AI*, 5(4):1743–1758.
- AlKhafaji, B. J., Salih, M. A., Shnain, S., and Nabat, Z. (2020). Segmenting video frame images using genetic algorithms. *Periodicals of Engineering and Natural Sciences (PEN)*, 8(2):1106–1114.
- Alsaeedi, A. and Khan, M. Z. (2019). Software defect prediction using supervised machine learning and ensemble techniques: a comparative study. *Journal of Software Engineering and Applications*, 12(5):85–100.
- Ayon, S. I. (2019). Neural network based software defect prediction using genetic algorithm and particle swarm optimization. In *2019 1st International Conference on Advances in Science, Engineering and Robotics Technology (ICASERT)*, pages 1–4.

- Bandyopadhyay, S., Murthy, C. A., and Pal, S. K. (1995). Pattern classification with genetic algorithms. *Pattern recognition letters*, 16(8):801–808.
- Bassel, A., Abed, S. A., Abdullah, S., Nordin, M. J., and Turkey, A. (2024). An improved glowworm swarm optimization based on various mutation operators. *IEEE Access*.
- Beja-Battais, P. (2023). Overview of adaboost: Reconciling its views to better understand its dynamics. *arXiv preprint arXiv:2310.18323*.
- Bernard, T., Gnauck, A., Jacobi, M., Karimanzira, D., Krol, O., Pfützenreuter, T., Scharaw, B., and Westerhoff, T. (2015). *Modeling, Control and Optimization of Water Systems: Systems Engineering Methods for Control and Decision Making Tasks*. Springer.
- Beume, N., Naujoks, B., and Emmerich, M. (2007). Sms-emoa: Multiobjective selection based on dominated hypervolume. *European Journal of Operational Research*, 181(3):1653–1669.
- Boughorbel, S., Jarray, F., and El-Anbari, M. (2017). Optimal classifier for imbalanced data using matthews correlation coefficient metric. *PloS one*, 12(6):e0177678.
- Bowling, M., Fürnkranz, J., Graepel, T., and Musick, R. (2006). Machine learning and games. *Machine learning*, 63(3):211–215.
- Brady, F. (2013). Cambridge university study states software bugs cost economy \$312 billion per year.
- Casajus, A. (2019). Relaxations of symmetry and the weighted shapley values. *Economics letters*, 176:75–78.
- Chacón, J. and Segura, C. (2018). Analysis and enhancement of simulated binary crossover. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE.
- Chen, L., Guo, G., and Wang, K. (2011). Class-dependent projection based method for text categorization. *Pattern Recognition Letters*, 32(10):1493–1501.
- Chicco, D. and Jurman, G. (2020). The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC genomics*, 21:1–13.
- Chinenye, O., Anyachebelu, K. T., and Abdullahi, M. U. (2023). Software defect prediction system based on decision tree algorithm. *Asian Journal of Research in Computer Science*, 16(4):32–48.
- Cui, M., Long, S., Jiang, Y., and Na, X. (2022). Research of software defect prediction model based on complex network and graph neural network. *Entropy*, 24(10):1373.
- Damia, A., Esnaashari, M., and Parvizimosaed, M. (2021). Adaptive genetic algorithm based on mutation and crossover and selection probabilities. In *2021 7th International Conference on Web Research (ICWR)*, pages 86–90.

- Dasha, P. (2023). A comparative review of approaches for the evolutionary search to prevent premature convergence in ga. *Applied Soft Computing*, 25:1047–1077.
- Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197.
- Deep, K. and Mebrahtu, H. (2011). Combined mutation operators of genetic algorithm for the travelling salesman problem. *International Journal of Combinatorial Optimization Problems and Informatics*, 2(3):1–23.
- Deepa, S. N. and Sivanandam, S. N. (2010). *Introduction to Genetic Algorithms*. Springer, Berlin, Germany.
- Dipa, W. A. and Sunindyo, W. D. (2021). Software defect prediction using smote and artificial neural network. In *2021 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–4.
- Eiben, A. E. and Smith, J. E. (2015). *Introduction to evolutionary computing*. Springer.
- Emmerich, M. et al. (2007). Evolutionary computation: A survey. *Evolutionary Computation*, 11(6):712–731.
- Eshelman, L. J. and Schaffer, J. D. (1993). Real-coded genetic algorithms and interval-schemata. In *Foundations of genetic algorithms*, volume 2, pages 187–202. Elsevier.
- Esteves, G., Figueiredo, E., Veloso, A., Viggiano, M., and Ziviani, N. (2020). Understanding machine learning software defect predictions. *Automated Software Engineering*, 27(3):369–392.
- Fernández, F. and Isasi, P. (2008). Local feature weighting in nearest prototype classification. *IEEE Transactions on Neural Networks*, 19(1):40–53.
- Fernando, K. R. M. and Tsokos, C. P. (2021). Dynamically weighted balanced loss: class imbalanced learning and confidence calibration of deep neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 33(7):2940–2951.
- Filzmoser, P., Maronna, R., and Werner, M. (2008). Outlier identification in high dimensions. *Computational statistics & data analysis*, 52(3):1694–1711.
- Gan, H., Huang, R., Luo, Z., Xi, X., and Gao, Y. (2018). On using supervised clustering analysis to improve classification performance. *Information Sciences*, 454:216–228.
- Goel, A. and Khanna, S. (2008). On the network coding advantage for wireless multicast in euclidean space. In *2008 International Conference on Information Processing in Sensor Networks (ipSN 2008)*, pages 64–69. IEEE.
- Goldani, M. (2024). Evaluating feature selection methods for macro-economic forecasting, applied for inflation indicator of iran. *arXiv preprint arXiv:2406.03742*.

- Goldberg, D. E., Deb, K., and Rawlins, G. J. E. (1991). Foundations of genetic algorithms. *Elsevier*, 1:69.
- Goyal, S. (2022). Effective software defect prediction using support vector machines (svms). *International Journal of System Assurance Engineering and Management*, 13(2):681–696.
- Gray, D., Bowes, D., Davey, N., Sun, Y., and Christianson, B. (2010). Software defect prediction using static code metrics underestimates defect-proneness. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7. IEEE.
- Haldar, S. and Capretz, L. F. (2024a). Feature importance in the context of traditional and just-in-time software defect prediction models. In *2024 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 818–822. IEEE.
- Haldar, S. and Capretz, L. F. (2024b). Interpretable software defect prediction from project effort and static code metrics. *Computers*, 13(2):52.
- Hammad, M., Alqaddoumi, A., and Al-Obaidy, H. (2019). Predicting software faults based on k-nearest neighbors classification. *International Journal of Computing and Digital Systems*, 8(5):462–467.
- Han, B., Qiao, L.-N., Chen, J.-L., Zhang, X.-D., Zhang, Y.-X., and Zhao, Y.-H. (2021). Geneticknn: A weighted knn approach supported by genetic algorithm for photometric redshift estimation of quasars. *Research in Astronomy and Astrophysics*, 21(1):017.
- Han, S. and Xiao, L. (2022). An improved adaptive genetic algorithm. In *SHS web of conferences*, volume 140, page 01044. EDP Sciences.
- Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.
- Hazra, S. and Santra, A. (2019). Short-range radar-based gesture recognition system using 3d cnn with triplet loss. *IEEE Access*, 7:125623–125633.
- He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.
- Hernández-Orallo, J., Flach, P., and Ferri Ramírez, C. (2012). A unified view of performance metrics: Translating threshold choice into expected classification loss. *Journal of Machine Learning Research*, 13:2813–2869.

- Herrera, F., Lozano, M., and Sánchez, A. M. (2002). Multiple crossover per couple with selection of the two best offspring: an experimental study with the blx-*alpha* crossover operator for real-coded genetic algorithms. In *Advances in Artificial Intelligence—IBERAMIA 2002: 8th Ibero-American Conference on AI Seville, Spain, November 12–15, 2002 Proceedings*, pages 143–152. Springer Berlin Heidelberg.
- Heuillet, A., Couthouis, F., and Díaz-Rodríguez, N. (2022). Collective explainable ai: Explaining cooperative strategies and agent contribution in multiagent reinforcement learning with shapley values. *IEEE Computational Intelligence Magazine*, 17(1):59–71.
- Hosseini, S., Turhan, B., and Gunarathna, D. (2017). A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering*, 45(2):111–147.
- Hua, Y., Liu, Q., Hao, K., and Jin, Y. (2021). A survey of evolutionary algorithms for multi-objective optimization problems with irregular pareto fronts. *IEEE/CAA Journal of Automatica Sinica*, 8(2):303–318.
- Ibili, H., Gungor, A. C., Maciejewski, M., Smajic, J., and Leuthold, J. (2022). Genetic algorithm based geometry optimization of terahertz plasmonic modulator antennas. In *2022 23rd International Conference on the Computation of Electromagnetic Fields (COMPUMAG)*, pages 1–4. IEEE.
- Iqbal, A., Aftab, S., Ali, U., Nawaz, Z., Sana, L., Ahmad, M., and Husen, A. (2019). Performance analysis of machine learning techniques on software defect prediction using nasa datasets. *International Journal of Advanced Computer Science and Applications*, 10(5).
- Jadhav, V., Devale, P., Jadhav, R., Molawade, M., Mohite, S., and Bidwe, R. V. (2023). Bug predictive models based on data analytics and soft computing techniques: A survey. In *2023 10th International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 822–831. IEEE.
- Jadhav, Y. and Barati Farimani, A. (2022). Dominant motion identification of multi-particle system using deep learning from video. *Neural Computing and Applications*, 34(20):18183–18193.
- Jalali Varnamkhasti, M. and Vali, M. (2023). A special mutation operator in the genetic algorithm for fixed point problems. *Journal of Function Spaces*, 2023(1):7714095.
- Jebari, K. and Madiafi, M. (2013). Selection methods for genetic algorithms. *International Journal of Emerging Sciences*, 3(4):333–344.
- Ji, H., Huang, S., Wu, Y., Hui, Z., and Zheng, C. (2019). A new weighted naive bayes method based on information diffusion for software defect prediction. *Software Quality Journal*, 27(3):923–968.

- Jiang, L., Duan, J., Zheng, R., Shen, H., Li, H., and Xu, J. (2023). Optimization and simulation of garment production line balance based on improved ga. *International Journal of Simulation Modelling*, 22(2):303–314.
- Jiang, L., Zhang, H., and Cai, Z. (2008). A novel bayes model: Hidden naive bayes. *IEEE Transactions on knowledge and data engineering*, 21(10):1361–1371.
- Katoch, S., Chauhan, S. S., and Kumar, V. (2021). A review on genetic algorithm: past, present, and future. *Multimedia tools and applications*, 80:8091–8126.
- Kelly Jr, J. D. and Davis, L. (1991). A hybrid genetic algorithm for classification. In *IJCAI*, volume 91, pages 645–650.
- Keshavarz, H. and Nagappan, M. (2022). Apachejit: A large dataset for just-in-time defect prediction.
- Keshavarz, H. and Rodríguez-Pérez, G. (2024). Jitgnn: A deep graph neural network framework for just-in-time bug prediction. *Journal of Systems and Software*, 210:111984.
- Khalid, A., Badshah, G., Ayub, N., Shiraz, M., and Ghouse, M. (2023). Software defect prediction analysis using machine learning techniques. *Sustainability*, 15(6):5517.
- Khan, M. Z. (2020). Hybrid ensemble learning technique for software defect prediction. *International Journal of Modern Education & Computer Science*, 12(1):1–10.
- Khoshgoftaar, T. M., Allen, E. B., and Deng, J. (2002). Using regression trees to classify fault-prone software modules. *IEEE Transactions on reliability*, 51(4):455–462.
- Khoshgoftaar, T. M., Allen, E. B., Halstead, R., Trio, G. P., and Flass, R. M. (1998). Using process history to predict software quality. *Computer*, 31:66–72.
- Khoshgoftaar, T. M., Allen, E. B., Hudepohl, J. P., and Aud, S. J. (1997). Application of neural networks to software quality modeling of a very large telecommunications system. *IEEE Transactions on Neural Networks*, 8(4):902–909.
- Khoshgoftaar, T. M. and Szabo, R. M. (1996). Using neural networks to predict software faults during testing. *IEEE Transactions on Reliability*, 45(3):456–462.
- Köksoy, O. and Yalcinoz, T. (2008). Robust design using pareto type optimization: a genetic algorithm with arithmetic crossover. *Computers & Industrial Engineering*, 55(1):208–218.
- Kondo, M., German, D. M., Mizuno, O., and Choi, E.-H. (2020). The impact of context metrics on just-in-time defect prediction. *Empirical software engineering*, 25:890–939.
- Kulamala, V. K., Kumar, L., and Mohapatra, D. P. (2021). Software fault prediction using lssvm with different kernel functions. *Arabian Journal for Science and Engineering*, 46(9):8655–8664.

- Kumar, R. and Jyotishree (2012). Effect of polygamy with selection in genetic algorithm. *International Journal of Soft Computing and Engineering (IJSCE)*, 2(1):2231–2307.
- Kumar, R. S. and Sathyanarayana, B. (2015). Adaptive genetic algorithm based artificial neural network for software defect prediction. *Global Journal of Computer Science and Technology*, 15(1):23–32.
- Kuncheva, L. I. and Rodríguez, J. J. (2014). A weighted voting framework for classifiers ensembles. *Knowledge and information systems*, 38:259–275.
- Lambora, A., Gupta, K., and Chopra, K. (2019). Genetic algorithm-a literature review. In *2019 international conference on machine learning, big data, cloud and parallel computing (COMITCon)*, pages 380–384. IEEE.
- Le, P. and Zuidema, W. (2016). Quantifying the vanishing gradient and long distance dependency problem in recursive neural networks and recursive lstms. *arXiv preprint arXiv:1603.00423*.
- Lessmann, S., Baesens, B., Mues, C., and Pietsch, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE transactions on software engineering*, 34(4):485–496.
- Li, C. H., Gondra, I., and Liu, L. (2012). An efficient parallel neural network-based multi-instance learning algorithm. *The Journal of Supercomputing*, 62:724–740.
- Li, J., He, P., Zhu, J., and Lyu, M. R. (2017). Software defect prediction via convolutional neural network. In *2017 IEEE international conference on software quality, reliability and security (QRS)*, pages 318–328. IEEE.
- Li, S., Zhang, K., Chen, Q., Wang, S., and Zhang, S. (2020). Feature selection for high dimensional data using weighted k-nearest neighbors and genetic algorithm. *IEEE Access*, 8:139512–139528.
- Lim, S. M., Sultan, A. B. M., Sulaiman, M. N., Mustapha, A., and Leong, K. Y. (2017). Crossover and mutation operators of genetic algorithms. *International journal of machine learning and computing*, 7(1):9–12.
- Ling, M. H. (2023). An artificial life simulation library based on genetic algorithm, 3-character genetic code and biological hierarchy. *arXiv preprint arXiv:2304.13520*.
- Liu, Y., Li, Y., Guo, J., Zhou, Y., and Xu, B. (2018). Connecting software metrics across versions to predict defects. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 232–243. IEEE.
- Liu, Z., Liu, A., Wang, C., and Niu, Z. (2004). Evolving neural network using real coded genetic algorithm (ga) for multispectral image classification. *Future Generation Computer Systems*, 20(7):1119–1129.

- Lu, Y. and Salem, F. M. (2017). Simplified gating in long short-term memory (lstm) recurrent neural networks. In *2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS)*, pages 1601–1604. IEEE.
- Lundberg, S. (2017). A unified approach to interpreting model predictions. *arXiv preprint arXiv:1705.07874*.
- Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504–518.
- Malkauthekar, M. (2013). Analysis of euclidean distance and manhattan distance measure in face recognition. In *Third International Conference on Computational Intelligence and Information Technology (CIIT 2013)*, pages 503–507. IET.
- Manjula, C. and Florence, L. (2019). Deep neural network based hybrid approach for software defect prediction using software metrics. *Cluster Computing*, 22(Suppl 4):9847–9863.
- Mateos-García, D., García-Gutiérrez, J., and Riquelme-Santos, J. C. (2012). On the evolutionary optimization of k-nn by label-dependent feature weighting. *Pattern Recognition Letters*, 33(16):2232–2238.
- Mateos-García, D., García-Gutiérrez, J., and Riquelme-Santos, J. C. (2016). An evolutionary voting for k-nearest neighbours. *Expert Systems with Applications*, 43:9–14.
- Mateos-García, D., García-Gutiérrez, J., and Riquelme-Santos, J. C. (2019). On the evolutionary weighting of neighbours and features in the k-nearest neighbour rule. *Neurocomputing*, 326:54–60.
- Mathew, T. V. (2012). Genetic algorithm. *Report submitted at IIT Bombay*, 53.
- Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., and Bener, A. (2010). Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17:375–407.
- Merigó, J. M. and Casanovas, M. (2011). A new minkowski distance based on induced aggregation operators. *International Journal of Computational Intelligence Systems*, 4(2):123–133.
- Messaoud, S., Bradai, A., Bukhari, S. H. R., Quang, P. T. A., Ahmed, O. B., and Atri, M. (2020). A survey on machine learning in internet of things: Algorithms, strategies, and applications. *Internet of Things*, 12:100314.
- Michalewicz, Z. (2013). *Genetic algorithms+ data structures= evolution programs*. Springer Science & Business Media.
- Molnar, C. (2020). *Interpretable machine learning*. Lulu. com.
- Mullick, S. (2021). *On Class Imbalanced Learning: Design of Non-parametric Classifiers, Performance Indices, and Deep Oversampling Strategies*. PhD thesis, Indian Statistical Institute-Kolkata.

- Nawara, D. and Kashef, R. (2021). Deploying different clustering techniques on a collaborative-based movie recommender. In *2021 IEEE International Systems Conference (SysCon)*, pages 1–6. IEEE.
- Obitko, M. (1998). Introduction to genetic algorithms. *Czech Technical University: Prague, Czech Republic*.
- O’Neill, D., Lensen, A., Xue, B., and Zhang, M. (2018). Particle swarm optimisation for feature selection and weighting in high-dimensional clustering. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE.
- Owais, W. B., Alkhazendar, I. W., and Saleh, D. M. (2020). Evaluating the impact of different types of crossover and selection methods on the convergence of 0/1 knapsack using genetic algorithm. *arXiv preprint arXiv:2010.03483*.
- Oyewole, G. J. and Thopil, G. A. (2023). Data clustering: application and trends. *Artificial Intelligence Review*, 56(8):6439–6475.
- Pachouly, J., Ahirrao, S., Kotecha, K., Selvachandran, G., and Abraham, A. (2022). A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools. *Engineering Applications of Artificial Intelligence*, 111:104773.
- Pan, L., Gong, Z., Zhang, J., and Zhang, H. (2021). Adaptive simulated binary crossover for rotated multi-objective optimization. *Swarm and Evolutionary Computation*, 60:100759.
- Panda, B. (2019). A survey on application of population based algorithm on hyperparameter selection. *Department of Computer Science: Course*, 761:1–9.
- pandas development team, T. (2020). pandas-dev/pandas: Pandas.
- Paredes, R. and Vidal, E. (2006). Learning weighted metrics to minimize nearest-neighbor classification error. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(7):1100–1110.
- Park, D.-C. (2009). Centroid neural network with weighted features. *Journal of Circuits, Systems, and Computers*, 18(08):1353–1367.
- Peterson, L. E. (2009). K-nearest neighbor. *Scholarpedia*, 4(2):1883.
- Pinazo, M. A. (2024). New genetic gray wolf optimizer with a random selective mutation for wind farm layout optimization. *Heliyon*.
- Pornprasit, C. and Tantithamthavorn, C. K. (2021). Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 369–379. IEEE.
- Powers, D. M. (2020). Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *arXiv preprint arXiv:2010.16061*.

- Prakash, V. A., Ashoka, D., and Aradya, V. M. (2015). Application of data mining techniques for defect detection and classification. In *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014: Volume 1*, pages 387–395. Springer.
- Qiao, Y., Gong, L., Zhao, Y., Wang, Y., and Wei, M. (2024). Demuvgn: Effective software defect prediction model by learning multi-view software dependency via graph neural networks. *arXiv preprint arXiv:2410.19550*.
- Qiu, S., Wang, S., Tian, X., Huang, M., and Huang, Q. (2022). Visualization-based software defect prediction via convolutional neural network with global self-attention. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pages 189–198. IEEE.
- Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA.
- Rajakumar, B. (2013). Static and adaptive mutation techniques for genetic algorithm: a systematic comparative analysis. *International Journal of Computational Science and Engineering*, 8(2):180–193.
- Razali, N. M. and Geraghty, J. (2011). Genetic algorithm performance with different selection strategies in solving tsp. In *Proceedings of the World Congress on Engineering*, volume 2, Hong Kong, China. International Association of Engineers.
- Rodriguez-Perez, G., Nagappan, M., and Robles, G. (2020). Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the openstack project. *IEEE Transactions on Software Engineering*, 48(4):1400–1416.
- Rodríguez-Pérez, G., Robles, G., Serebrenik, A., Zaidman, A., Germán, D. M., and Gonzalez-Barahona, J. M. (2020). How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering*, 25:1294–1340.
- Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.
- Saadaoui, D., Elyaqouti, M., Assalaou, K., Lidaighbi, S., et al. (2021). Parameters optimization of solar pv cell/module using genetic algorithm based on non-uniform mutation. *Energy Conversion and Management: X*, 12:100129.
- Salgotra, R. and Mirjalili, S. (2024). Multi-algorithm based evolutionary strategy with adaptive mutation mechanism for constraint engineering design problems. *Expert Systems with Applications*, 258:125055.
- Schuh, M. A., Wylie, T., and Angryk, R. A. (2014). Mitigating the curse of dimensionality for exact knn retrieval. In *The Twenty-Seventh International Flairs Conference*.
- Serpell, M. and Smith, J. E. (2010). Self-adaptation of mutation operator and probability for permutation representations in genetic algorithms. *Evolutionary Computation*, 18(3):491–514.

- Shi, H., Zhang, X., Gao, Y., Wang, S., and Ning, Y. (2023). Robust total least squares estimation method for uncertain linear regression model. *Mathematics*, 11(20):4354.
- Šikić, L., Afrić, P., Kurdija, A. S., and Šilić, M. (2021). Improving software defect prediction by aggregated change metrics. *IEEE access*, 9:19391–19411.
- Sivanandam, S. N. and Deepa, S. N. (2008). *Introduction to Genetic Algorithm*. Springer-Verlag, Berlin, Heidelberg, 1st edition.
- Skorikov, M. and Momen, S. (2020). Machine learning approach to predicting the acceptance of academic papers. In *2020 IEEE International Conference on Industry 4.0, Artificial Intelligence, and Communications Technology (IAICT)*, pages 113–117.
- Śliwerski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5.
- Song, J.-L., Tan, Y.-H., and Xiong, Z.-T. (2013). Multilevel parallel quantum-behaved particle swarm optimization based on asymmetrical boundary variation. *Jisuanji Yingyong Yanjiu*, 30(6).
- Song, S., Jin, H., and Yang, Q. (2021). Performance analysis of different operators in genetic algorithm for solving continuous and discrete optimization problems. In *ICEIS (1)*, pages 536–547.
- Squires, G. L. (2001). *Practical physics*. Cambridge university press.
- Srinivas, M. and Patnaik, L. M. (1994). Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(4):656–667.
- Srivastava, N. (2013). Improving neural networks with dropout. Master’s thesis, University of Toronto.
- Su, M.-Y., Chang, K.-C., Wei, H.-F., and Lin, C.-Y. (2008). Feature weighting and selection for a real-time network intrusion detection system based on ga with knn. In Yang, C. C., Chen, H., Chau, M., Chang, K., Lang, S.-D., Chen, P. S., Hsieh, R., Zeng, D., Wang, F.-Y., Carley, K., Mao, W., and Zhan, J., editors, *Intelligence and Security Informatics*, pages 195–204, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Sun, Y., Xue, B., Zhang, M., Yen, G. G., and Lv, J. (2020). Automatically designing cnn architectures using the genetic algorithm for image classification. *IEEE transactions on cybernetics*, 50(9):3840–3854.
- Sundararajan, M. and Najmi, A. (2020). The many shapley values for model explanation. In *International conference on machine learning*, pages 9269–9278. PMLR.
- Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. Technical Report NIST Planning Report 02-3, National Institute of Standards and Technology, Gaithersburg, MD.

- Thota, M. K., Shajin, F. H., Rajesh, P., et al. (2020). Survey on software defect prediction techniques. *International Journal of Applied Science and Engineering*, 17(4):331–344.
- Tian, Y., Si, L., Zhang, X., Cheng, R., He, C., Tan, K. C., and Jin, Y. (2021). Evolutionary large-scale multi-objective optimization: A survey. *ACM Computing Surveys (CSUR)*, 54(8):1–34.
- Toma, R. N., Prosvirin, A. E., and Kim, J.-M. (2020). Bearing fault diagnosis of induction motors using a genetic algorithm and machine learning classifiers. *Sensors*, 20(7):1884.
- Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. (2017). Graph attention networks. *arXiv preprint arXiv:1710.10903*.
- Venkateswarlu, B. and Gangula, R. (2024). Exploring the power and practical applications of k-nearest neighbours (knn) in machine learning. *Journal of Computer Allied Intelligence (JCAI, ISSN: 2584-2676)*, 2(1):8–15.
- Vouillon, J. and Di Cosmo, R. (2013). Broken sets in software repository evolution. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 412–421.
- Wahono, R. S., Herman, N. S., and Ahmad, S. (2014). Neural network parameter optimization based on genetic algorithm for software defect prediction. *Advanced Science Letters*, 20(10-11):1951–1955.
- Wang, H., Zhuang, W., and Zhang, X. (2021a). Software defect prediction based on gated hierarchical lstms. *IEEE Transactions on Reliability*, 70(2):711–727.
- Wang, K., Liu, L., Yuan, C., and Wang, Z. (2021b). Software defect prediction model based on lasso-svm. *Neural Computing and Applications*, 33:8249–8259.
- Wang, X. (2023). The improvement and comparison study of distance metrics for machine learning algorithms for indoor wi-fi localization. *IEEE Access*.
- Wang, X. and Yu, X. (2024). Differential evolution algorithm with three mutation operators for global optimization. *Mathematics*, 12(15):2311.
- Wes McKinney (2010). Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61.
- Wu, N., Zhao, X. W., Wang, J., and Pan, D. (2020a). Learning effective road network representation with hierarchical graph neural networks. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 6–14.
- Wu, S., Zhang, G., Eshraghian, J. K., Chua, L. O., and Basu, A. (2018). l_1 -norm batch normalization for efficient training of deep neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 30(7):2043–2051.

- Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Philip, S. Y. (2020b). A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24.
- Xu, H., Zeng, W., Zeng, X., and Yen, G. G. (2018). An evolutionary algorithm based on minkowski distance for many-objective optimization. *IEEE transactions on cybernetics*, 49(11):3968–3979.
- Xue, Y., Zhu, H., Liang, J., and Słowiak, A. (2021). Adaptive crossover operator based multi-objective binary genetic algorithm for feature selection in classification. *Knowledge-Based Systems*, 227:107218.
- Yadav, S. L. and Sohal, A. (2017). Comparative study of different selection techniques in genetic algorithm. *International Journal of Engineering, Science and Mathematics*, 6(3):174–180.
- Yang, C., Qian, Q., Wang, F., and Sun, M. (2016). An improved adaptive genetic algorithm for function optimization. In *2016 IEEE International Conference on Information and Automation (ICIA)*, pages 675–680. IEEE.
- Yang, D., Yu, Z., Yuan, H., and Cui, Y. (2022). An improved genetic algorithm and its application in neural network adversarial attack. *Plos one*, 17(5):e0267970.
- Yang, L., Zhou, Y., Fu, H., Liu, M.-Q., and Zheng, W. (2024). Fast approximation of the shapley values based on order-of-addition experimental designs. *Journal of the American Statistical Association*, 119(547):2294–2304.
- Yusoff, M. and Othman, A. A. (2018). Genetic algorithm with elitist-tournament for clashes-free slots of lecturer timetabling problem. *Indonesian Journal of Electrical Engineering and Computer Science*, 12(1):303.
- Zhang, J., Chung, H. S.-H., and Hu, B. (2004). Adaptive probabilities of crossover and mutation in genetic algorithms based on clustering technique. In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, volume 2, pages 2280–2287. IEEE.
- Zhang, J., Chung, H. S.-H., and Lo, W.-L. (2007). Clustering-based adaptive crossover and mutation probabilities for genetic algorithms. *IEEE Transactions on evolutionary Computation*, 11(3):326–335.
- Zhang, J. and Qu, S. (2021). Optimization of backpropagation neural network under the adaptive genetic algorithm. *Complexity*, 2021(1):1718234.
- Zhang, M.-L. and Zhou, Z.-H. (2005). A k-nearest neighbor based algorithm for multi-label classification. In *2005 IEEE international conference on granular computing*, volume 2, pages 718–721. IEEE.
- Zhang, S. (2021). Challenges in knn classification. *IEEE Transactions on Knowledge and Data Engineering*, 34(10):4663–4675.

- Zhao, Y., Ding, Y., and Pei, Y. (2024). Adaptive optimization in evolutionary reinforcement learning using evolutionary mutation rates. *IEEE Access*.
- Zhou, Y., Shan, C., Sun, S., Wei, S., and Zhang, S. (2019). Software defect prediction model based on kpca-svm. In *2019 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computing, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCCom/IOP/SCI)*, pages 1326–1332.
- Zou, Q., Lu, L., Qiu, S., Gu, X., and Cai, Z. (2021). Correlation feature and instance weights transfer learning for cross project software defect prediction. *IET Software*, 15(1):55–74.
- Ömer Faruk Arar and Ayan, K. (2017). A feature dependent naive bayes approach and its application to the software defect prediction problem. *Applied Soft Computing*, 59:197–209.