

SOFTWARE FAULT PREDICTION BASED ON SOURCE CODE

**SUMMARY OF THE DOCTORAL DISSERTATION IN
ENGINEERING**

Danang, 2026

Table of contents

Introduction.....	1
Chapter 1: Software Fault Prediction Overview.....	3
1.1. Concept and Process of Software Fault Prediction.....	3
1.2. Source Code Features and Selection of Features	3
1.3. Feature Extraction and Data Sampling	5
1.4. Predicting software faults based on source code semantic features	5
Chapter 2: Improving Software Fault Prediction with Wrapper Feature Selection Methods...	6
2.1. Introduction	6
2.2. Research methods.....	7
2.3. Experimental and analytical results	8
2.3.1. Effectiveness of Feature Selection Techniques (Research Question 1)	8
2.3.2. The Best Working Wrapper Method (Research Question 2).....	8
2.3.3. Results of statistical inspection	9
2.4. Chapter summary	9
Chapter 3: Combining feature selection and data sampling techniques in software fault prediction	10
3.1. Introduction	10
3.2. Proposed Method.....	11
3.3. Experimental and analytical results.....	11
3.3.1. Results of combining RFE and GAN models (Research Question 1).....	12
3.3.2. Results of Combining Autoencoders and GAN Models	12
3.3.3. The Most Effective GAN Method (Research Question 2)	13
3.3.4. Results of statistical evaluation	13
3.4. Chapter Summary.....	13
Chapter 4: Building a Transformer-Based Software Fault Prediction Model Using Syntax Tree and Word Embedding	15
4.1. Introduction	15
4.2. Related studies and research issues.....	16
4.3. Proposed Method	17

4.4. Implementation of experiments, results and analysis	17
4.4.1. Predicting Faults in Within-Project Defect Prediction	18
4.4.2. Predicting faults in Cross-Project Defect Prediction.....	18
4.4.3. Comparison of training time costs.....	19
4.4.4. Threats to the credibility and value of the research.....	19
4.5. Chapter summary	20
Conclusions and development directions	21
Conclusions	21
Future research directions	21

Introduction

Software plays an important role in many modern technology systems and applications, from defense and air traffic control to consumer devices. With the increase in the size and complexity of software, ensuring quality and reliability has become extremely important. However, due to budget and time constraints, it is impossible to create a completely fault-free software. Software bugs, even the smallest, can lead to skewed results and undesirable behavior of the program, causing project failure.

Software Fault Prediction (SFP) has been an important area of research for decades, aiming to identify software modules that are likely to contain bugs early in the development process. In this way, developers can focus their resources and testing efforts on potentially risky areas, thereby optimizing the development process, controlling costs, and improving software quality.

Software fault prediction focuses on designing and developing predictive models based on software data (e.g., source code complexity, change history, etc.) to predict faults in new pieces of code (i.e., files, classes, methods, etc.). These predictive models help software developers focus on fault-prone modules and optimize their efforts and resources. Over the years, machine learning techniques have been applied to the development of models that predict software faults. These techniques use historical software failure data (collected from previous software projects) to train the predictive model and identify faulty modules. Initially, fault prediction models used traditional machine learning techniques such as Naive Bayes, Support Vector Machine (SVM), and Decision Tree, which were trained on historical fault data and source code measurements. However, these methods often face some significant challenges:

1. **Irrelevant or Redundant Features:** Faulty datasets often have too many features, not all of which have a classification value. This redundancy increases computational costs and can reduce the performance of the model. Feature selection is necessary to eliminate unnecessary information, making the model learn faster and more accurately.
2. **Class Imbalance:** Software bug datasets are often severely unbalanced, with a much larger number of fault-free modules (majority classes) than faulty modules (minority classes). This causes machine learning models to tend to bias the majority class, resulting in unreliable predictions for faulty modules.
3. **Lack of Semantic and Syntactic Information:** Traditional fault prediction models often rely on manual measurements, failing to fully capture the syntactic and semantic structure of the source code. The two source code files have the same traditional

measurements but may differ semantically, affecting predictive performance. As a result, recent studies have turned to deep learning to automatically extract these features.

This study focuses on solving the above challenges through three main contributions: (1) On the basis of improving the efficiency of software fault prediction through the systematic evaluation and comparison of filter and wrapper characterization selection methods to determine the optimal feature set, contributing to reducing data dimensions, eliminating redundant and irrelevant features, thereby improving accuracy and providing a test base to help select the right method for the problem of predicting software faults; (2) Propose and evaluate the experimental application of GAN antagonistic data generation models and their variants to solve the problem of data imbalance in tabular software fault datasets. Apply GAN models to generate high-quality synthetic data for the minority layer, thereby balancing the layer distribution, improving accuracy, and increasing the generalization of software fault prediction models. The results of the study provide an important testing basis for researchers to select the appropriate data generation method for machine learning problems with imbalanced data in the field of software engineering; (3) Propose a deep learning model based on the Transformer architecture, combined with semantic representation of the source code using the FastText embedding technique extracted from the abstract syntax tree. The model is designed to automatically exploit and learn the latent syntax and semantic features in the source code, thereby improving the ability to identify and predict fault-prone modules. Unlike traditional methods based on manual characterization, the proposed model is capable of extracting meaningful semantic vector representations from the AST structure, while overcoming the limitations of sequential networks in modeling complex dependencies in the source code. By using *Transformer's self-attention* mechanism, the model can capture global contextual relationships, contributing to improved accuracy and generalization in software fault prediction.

Chapter 1: Software Fault Prediction Overview

This chapter provides an overview of software fault prediction, techniques for improving the performance of code-based fault prediction models, and related challenges.

1.1. Concept and Process of Software Fault Prediction

Software fault prediction focuses on building predictive models using historical software data (e.g., source code complexity, change history) to identify faults in new pieces of code (e.g., files, classes, methods). The goal is to help developers focus on fault-prone modules and optimize resources.

The software fault prediction process consists of the following key steps:

1. Data collection: Bug datasets are extracted from public data repositories such as PROMISE, NASA, Apache.
2. Data preprocessing: Noise treatment, irrelevant/redundant feature removal, outlier value processing, and data layer balancing to improve performance. Techniques such as data normalization, feature selection, characterization extraction, and data sampling are applied.
3. Feature extraction and training dataset generation: Features are extracted from source code or historical logs, and then combined with fault information to create a training dataset.
4. Build a model that predicts faults: Use statistical and machine learning techniques to build the model.
5. Performance evaluation: Evaluate the model using a test dataset, using measurements such as accuracy, precision, recall, F1-score, and AUC.
6. Parameter Adjustment: Fine-tune the model parameters for the highest accuracy and performance.
7. Fault prediction: Apply a trained model to identify faulty modules in real-world projects.

1.2. Source Code Features and Selection of Features

A key element of any engineering process is measurement. The measurements are used to better understand the properties of the model we build. More importantly, they are used to evaluate the quality of a technical product or the process by which it is created. Unlike other engineering disciplines, software engineering is not based on the basic quantitative laws of physics. Absolute measurements such as voltage, mass, velocity or temperature are rarely available in the software sector. Instead, we seek to construct a set of indirect measurements

to form metrics that reflect the quality of some form of software representation. Recognizing the importance of software measurements, there have been many proposed measurements for measurement. These measurements try to capture different aspects of the software product as well as the software development process. Several degrees are used to measure the same aspect of the software; For example, there are many proposed measurements to measure the degree of coupling between different layers. Therefore, software developers need to specify the relationship between different degrees of measurement and measurement of a software aspect. For example, if we want to know the size of a table, there can be many measurements involved such as the length, width, area, or diagonal of the table. However, the length and width measurements are sufficient, as other measurements can be inferred from them as needed. Similarly in the field of software, it is necessary to determine the measurements that are really necessary and provide useful information; otherwise, the manager will be confused by too many numbers and the target of using the measurement will be skewed. When comparing the number of existing measurements with the number of core features they actually reflect, a significant degree of redundancy can be observed. Code Metrics are indirect measurements that reflect the quality of the software. Many measurements have been proposed to measure various aspects such as coupling, source code complexity, and line of code (LOC). However, having too many measurements can lead to information redundancy.

Feature selection is an important technique for reducing data dimensionality by eliminating irrelevant or redundant features, which speeds up model training and improves accuracy. There are four main types of feature selection methods:

- Filter: Feature evaluation based on intrinsic attributes (information, dependency, consistency, distance), independent of machine learning algorithms.
- Wrapper: Selects the optimal set of features based on the performance of a specific machine learning algorithm, by minimizing classifier estimation faults. This method is usually more computationally time-consuming than filters.
- Embedded: Integrates a featured selection process into the learning algorithm, which has higher computational performance than wrappers.
- Hybrid: Combine two or more different methods to take advantage of complementary advantages.
- In addition, metaheuristic algorithms are also widely used in feature selection, although it is not guaranteed to find the most optimal combination due to its random nature.

1.3. Feature Extraction and Data Sampling

Feature Extraction is a technique that transforms an input set of features into a new, less redundant, and more relevant set of features, in order to reduce complexity and provide a more understandable representation. Common methods include Key Component Analysis (PCA) and Linear Differential Analysis (LDA).

Data Sampling solves the problem of layer imbalance by adjusting the distribution of data.

- **Oversampling:** Creating new composite samples for the minority class to increase the number of samples. Common techniques are SMOTE and variants (Borderline-SMOTE, ADASYN). However, they can lead to overfitting due to synthetic samples that are too similar to the existing one.
- **Undersampling:** Removes some samples of the majority class to balance the ratio. The most common technique is Random Under-Sampling (RUS). The downside is that important information can be lost. In software fault prediction, oversampling is often preferred over undersampling.

1.4. Predicting software faults based on source code semantic features

Deep learning models, built on artificial neural networks, have been widely used in software fault prediction to exploit semantic and syntactic information of source code. Instead of relying on manual measurements, these models extract specific nodes from the source code's Abstract Syntax Tree (AST) and convert them into strings/vectors for inclusion in the deep learning model. Popular deep learning models include the Deep Belief Network (DBN), Convolutional Neural Network (CNN), Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU).

However, sequential networks such as RNNs and LSTMs still have limitations in fully representing syntactic and semantic information from AST, as well as having difficulty grasping complex dependencies in the source code. At the same time, problems such as vanishing gradient and exploding gradient can still occur.

Chapter 2: Improving Software Fault Prediction with Wrapper Feature Selection Methods

This chapter examines the utilization of wrapper feature selection methods to reduce data redundancy and improve the performance of the software fault prediction model.

2.1. Introduction

Feature selection is an important step in the process of analyzing data and applying machine learning models. The goal of these methods is to identify the set of features or variables that have the highest degree of relevance to the target variable. Classification models trained on a collapsed feature space typically have higher stability and reproducibility than models built on the entire original feature space of large size. The featured selection technique improves model performance by reducing computational complexity and increasing the interpretability of model results. Featured selection methods play a key role in the process of building a machine learning model. In the process of feature selection, the main goal is to look for significant traits or highly correlated traits. Features that do not provide useful information are called irrelevant features, while features that do not carry new information compared to the features that have been selected are called redundant features. In addition, features that have no relationship or correlation with the target variable are considered noise. The existence of noise will distort the prediction process, which in turn reduces the classification performance. Therefore, noise processing is necessary to improve predictive efficiency, and this can be achieved through data dimensional reduction [54]. A better feature selection algorithm should provide many benefits such as providing insight into the data, building a more efficient classification model, enhancing generalization, and identifying irrelevant features. Besides, it also helps to better understand the relationship between features and target variables, reducing computational requirements in the process of solving a specific problem. In datasets with a large number of dimensions, where the number of attributes outweighs the number of samples, selecting the appropriate characterization reduces the dimension of the data, improves model performance, and saves costs and computational time. The primary goal of wrapper feature selection is to extract a subset of the most relevant features from the original dataset in order to reduce the number of dimensions while maintaining or improving classification performance. Wrapper methods often give results that are superior to filter methods. This study empirically evaluated ten different metaheuristic algorithms in wrapper methods to reduce data redundancy in fault prediction datasets.

2.2. Research methods

The study used nine software bug datasets from the PROMISE and AEEEM repositories, including independent variables such as Lines of Code (LOCode), Lines of Comments (LOComment), and dependent (fault/no fault) variables. These datasets are characterized by a predominantly fault-free sample ratio compared to the fault sample, causing the problem of data imbalance.

Ten methods of selecting a feature wrapper based on metaheuristic were surveyed:

- Artificial Butterfly Optimization (ABO)
- Atom Search Optimization (ASO)
- Equilibrium Optimizer (EO)
- Henry Gas Solubility Optimization (HGSO)
- Poor and Rich Optimization (PRO)
- Generalized Normal Distribution Optimization (GNDO)
- Slime Mould Algorithm (SMA)
- Harris Hawks Optimization (HHO)
- Path Finder Algorithm (PFA)
- Manta Ray Foraging Optimization (MRFO)
- Recursive Feature Elimination (RFE)

These methods are compared to the baseline model, in which machine learning algorithms are applied directly to the original data without feature selection.

Three machine learning models are used to evaluate classification performance:

- Random Forest (RF): Combines multiple decision trees, renowned for its high accuracy and resistance to overfitting.
- AdaBoost: Improve weak classifiers by sequentially training multiple classifiers and adjusting weights.
- Extra Trees (ET): A variation of RF, in which attribute selection is done randomly, which speeds up training and reduces overfitting.

Performance measures include Precision, Recall, F1-score, and Area Under the Curve (AUC). The Wilcoxon signed-rank test is used to determine the statistical significance of performance differences between techniques at a significance level of 0.05. Each experiment is repeated ten times to ensure reliability.

2.3. Experimental and analytical results

The study addresses two main questions:

1. How effectively are feature selection techniques applied in improving the ability to predict software failures by eliminating irrelevant or redundant features?
2. Which wrapper feature selection method works best in selecting the optimal features for software fault prediction?

2.3.1. Effectiveness of Feature Selection Techniques (Research Question 1)

Experimental results show that all wrapper feature selection methods are generally superior to the base model (no feature selection applied). This demonstrates that the elimination of irrelevant and redundant features is an important step in improving software fault prediction performance.

- Random Forest: Recursive Feature Elimination (RFE) combined with Random Forest achieved the highest average performance across all measurements (Precision, Recall, AUC, F1-score), followed by Equilibrium Optimizer (EO), Poor and Rich Optimization (PRO).
- With AdaBoost: Henry Gas Solubility Optimization (HGSO) consistently achieves the highest Precision and AUC values across the majority of datasets. EO achieved the highest average Recall and F1-score values.
- With Extra Trees: RFE outperforms other methods in terms of Recall, F1-score, and average AUC. ASO achieves the highest average Precision value.

2.3.2. The Best Working Wrapper Method (Research Question 2)

Based on AUC, RFE, EO and PRO values consistently deliver superior results across multiple data sets. Specifically, RFE achieved the highest AUC values on KC1, EQ, Lucene, PDE. The PRO also achieved notable AUC on PC1, EQ, JDT.

RFE combined with Random Forest achieved the highest average AUC score (0.87), EO and Random Forest, followed by PRO combined with Random Forest (0.84). This confirms that RFE is the most effective feature selection method for predicting software faults, which improves the performance of classifiers. RFE, EO, PRO, and HGSO also exhibit strong and stable performance.

2.3.3. Results of statistical inspection

The Wilcoxon signed-rank test at a significance level of $\alpha = 0.05$ was performed to compare the wrapper methods with the baseline model.

- RFE, ASO, HGSO, PRO, PFA and MRFO achieved statistically significant improvements in F1-scores compared to the baseline method when using Random Forest.
- For example, ASO has a P-value of 0.026 and an effect size $r = 0.913$, with an average difference of 0.071.
- EO shows the average r -size effect but the P-value is not statistically significant (0.293).
- SMA and HHO have no statistically significant differences.

This emphasizes that not every difference in numerical values is statistically significant. Overall, RFE, ASO, HGSO, PRO, PFA, and MRFO are recommended for implementation in real-world fault prediction models due to their efficiency and statistical significance.

2.4. Chapter summary

Chapter 2 demonstrated that the elimination of irrelevant and redundant features using wrapper feature selection methods is essential for building a robust software fault prediction model. The wrapper methods were evenly evaluated for better performance than the base model. In particular, RFE is the most effective method, followed by RFE, EO, PRO, and HGSO. The sum of the empirical results shows that a combination of feature selection and appropriate data balancing techniques can significantly improve the performance of software fault prediction models, both in terms of accuracy and stability.

However, this study still has some limitations: the datasets used are static and relatively balanced, which may not fully reflect real-world situations. Evaluating only three classifiers can also limit generalization. In the future, the study will integrate wrapper feature selection methods with sampling techniques (such as SMOTE, ADASYN) to solve the problem of unbalanced and high-dimensional data.

Chapter 3: Combining feature selection and data sampling techniques in software fault prediction

This chapter focuses on data sampling models to address data imbalances, thereby improving the accuracy and efficiency of fault prediction models.

3.1. Introduction

To develop models that predict software failures, many machine learning techniques have been studied and shown promising results on software failure datasets. During training, the predictive model learns the features of previous software projects and makes predictions about whether a new module is likely to fail. However, in many cases, the number of modules with faults in the training dataset is very small compared to the number of modules without faults, resulting in a data imbalance issue. This results in predictive models built on unbalanced software datasets that are often unable to learn the patterns feature of fault modules and therefore tend to mispredict multiple fault modules. As a result, predictive models become unreliable and difficult to apply in practice. Therefore, proposing techniques or methods to overcome the problem of layer imbalance in fault datasets has been considered an important issue in the field of software fault prediction. Many previous studies aimed at developing different methods to address this class imbalance. Data sampling techniques such as oversampling and undersampling are used to balance the data. However, traditional oversampling (such as SMOTE) can lead to overfitting due to synthetic samples being too similar, while undersampling can lose important information.

Generative Adversarial Networks (GANs) have recently been successfully applied to address data imbalances in a variety of areas, including photo and video generation, data augmentation, and natural language processing. Existing resampling techniques such as SMOTE rely heavily on local information (i.e., neighboring data points) to generate synthetic data samples. However, the GAN method has the ability to learn the overall distribution of data across different features as well as the correlation between them. As a result, GANs can produce synthetic data samples that are more realistic than traditional techniques. GANs were originally developed to generate images, tabular data, and other types of data. GAN-based approaches have been proven to learn data distribution better than traditional data sampling techniques and help effectively address data imbalances. GAN networks have recently been successfully applied to solve the problem of data imbalance. Unlike SMOTE, which relies solely on local information, GANs have the ability to learn the overall distribution of data across different features and correlations between them, helping to create more realistic and diverse synthetic patterns.

This study experimentally evaluated five variants of GAN to generate tabular software fault dataset: VanillaGAN, CTGAN, WGANGP, CopulaGAN, and TGAN. The synthetic samples generated by the GAN have a high similarity with the original data distribution of the minority class, which effectively solves the problem of imbalance.

The study used five fault datasets from the PROMISE repository, applied six machine learning algorithms (Random Forest, XGBoost, HGBost, MLP, Extra Trees, AdaBoost) and evaluated using Precision, Recall, F1-score, AUC, and MCC. The Wilcoxon test was also used to assess statistically significant differences. In addition, the performance between the GAN models and the base sampling models, including SMOTE, ADASYN, Borderline-SMOTE, and RUS, was compared to validate the effectiveness of synthetic data generation in predicting software failures.

3.2. Proposed Method

The research process consists of the following main steps:

1. Data Collection: Five fault datasets from PROMISE (CM1, KC1, KC2, PC1, JM1) are used, which are characterized by an unbalanced number of fault samples and no faults.
2. Data preprocessing: Fill in the missing values, normalize the data using z-normalization, and divide the data into training sets (70%) and tests (30%) using StratifiedKfold.
3. Feature selection and feature extraction:
 - Recursive Feature Elimination (RFE) was chosen as the most effective feature selection method. RFE eliminates the recursion of redundant and irrelevant features, which reduces data dimensionality and increases accuracy. Comparison results with ABO, ASO, HGSO, PRO show that RFE achieves the highest performance across all data sets.
 - Autoencoders are used for characterization extraction, based on research by Malhotra et al.
4. Data sampling by GAN: Five GAN variants (VanillaGAN, CTGAN, WGANGP, CopulaGAN, TGAN) are applied to the training subsets that have been characteristically processed (by RFE or Autoencoders) to produce balanced datasets.
5. Model Training and Evaluation: Balanced training datasets are fed into machine learning algorithms to build predictive models. These models were evaluated using Precision, Recall, F1-score, AUC, and MCC on the test set.

3.3. Experimental and analytical results

The study answers two key questions:

1. How effective are the GANs variations for data oversampling in handling imbalanced classes for SFP?
2. Which variant of GANs is superior among the state-of-the-art techniques handling imbalanced data for SFP?

3.3.1. Results of combining RFE and GAN models (Research Question 1)

- With Random Forest: WGANGP achieved the highest performance across all measurements, followed by VanillaGAN and CTGAN.
- With XGBoost: CTGAN showed the best performance, followed by WGANGP and VanillaGAN. WGANGP is prominent on PC1, while CTGAN is on CM1, KC1, KC2.
- With AdaBoost: CTGAN excels in recall and F1-score across most datasets (CM1, KC1, JM1). Best WGANGP on PC1. CopulaGAN achieves the highest precision on CM1, KC1, KC2.
- With HGBost: Similar to XGBoost, CTGAN has the best results, followed by WGANGP and VanillaGAN.
- With Multilayer Perceptron (MLP): GAN models all offer better performance. CopulaGAN achieves the highest precision on CM1, PC1, KC1, JM1. WGANGP achieved the highest AUC on PC1, KC2, JM1.
- With Extra Trees: CTGAN outperforms CM1, KC1, KC2, JM1 in terms of recall, F1-score, AUC, MCC. CopulaGAN achieves the best precision on the CM1 and KC1. Best WGANGP on PC1.

Overall, GAN methods (especially CTGAN, WGANGP, and VanillaGAN) offer better performance than traditional baseline sampling methods (SMOTE, ADASYN, Borderline-SMOTE, RUS). Fault prediction performance is improved by 1-4% compared to the base models.

3.3.2. Results of Combining Autoencoders and GAN Models

- With Random Forest: CopulaGAN achieved the highest average value in recall and F1-score, and the best in precision and MCC.
- With XGBoost: CopulaGAN achieves better performance on CM1, KC2, JM1.
- With AdaBoost: CopulaGAN achieves the highest precision, while CTGAN gives better results in recall and F1-score.
- With HGBost: CTGAN shows better performance on CM1, JM1, achieving the highest values in terms of recall, F1-score, average AUC.
- With Multilayer Perceptron (MLP): CTGAN also achieved the highest values in terms of recall, F1-score, average AUC.

- With Extra Trees: WGANGP achieved the highest values in terms of precision, recall, and average F1-score.

The results showed that the combination of Autoencoders and CTGAN delivered the best performance, followed by the pair of Autoencoders and CopulaGAN. Feature extraction using Autoencoders is efficient.

3.3.3. The Most Effective GAN Method (Research Question 2)

Figure 3.2 and analyses show that CTGAN combined with Extra Trees outperforms all other sampling methods in terms of mean AUC value (0.806). The proposed GAN models all have an average AUC value greater than 0.704. TGAN and CopulaGAN give similar results to SMOTE, ADASYN, RUS, and BorderMOTE.

Therefore, CTGAN is the most effective method for dealing with the problem of data imbalance in software fault prediction, followed by WGANGP and VanillaGAN. This is because the GAN architecture has the ability to learn the distribution of fault classes, producing more diverse and high-quality synthetic patterns.

3.3.4. Results of statistical evaluation

Wilcoxon testing (significance level $\alpha = 0.05$) was performed to evaluate GAN variants compared to traditional sampling methods.

- VanillaGAN has statistically significant performance differences compared to BorderSMOTE and RUS in terms of Recall, F1-score, and MCC.
- CTGAN differs statistically significantly from most methods (except SMOTE) on multiple measurements.
- WGANGP also has statistically significant differences compared to other methods.
- CopulaGAN and TGAN have statistically significant differences in Precision and MCC, but the mean may be lower at some other measurements.

These statistical tests confirm that CTGAN, WGANGP, and VanillaGAN deliver better predictive performance and are superior to traditional data sampling methods.

3.4. Chapter Summary

Chapter 3 demonstrated that redundant features and class imbalances are the main factors influencing fault prediction model performance. The thesis proposes an approach using variants of GAN antagonistic bionets including VanillaGAN, CTGAN, WGANGP, CopulaGAN, and TGAN for tabular software bug data synthesis. In addition, to select an

optimal feature set, we use the RFE feature selection technique and feature extraction Autoencoders on the unbalanced dataset, and then apply GAN variations to generate new patterns for the minority class to balance the dataset. Next, we examine the performance of the machine learning algorithms on a balanced dataset (Random Forest, XGBoost, MLP, AdaBoost, Extra Trees, and HistGradientBoosting), using the optimal feature set extracted from the previous steps. We conducted tests on five software bug datasets and used evaluation indicators such as precision, recall, F1-score, AUC, and MCC to evaluate the effectiveness of GAN models in sampling additional data. In addition, this study performed a comparative analysis between the performance of GAN variants and traditional data sampling methods. By combining the RFE feature selection technique/Autoencoders feature extraction with GAN variants, the study significantly improved the performance of machine learning models in an unbalanced data environment. CTGAN, WGAN-GP, and VanillaGAN are the most effective sampling methods. In the future, these methods can be extended to the problem of predicting faults between projects and other programming languages.

Chapter 4: Building a Transformer-Based Software Fault Prediction Model Using Syntax Tree and Word Embedding

This chapter examines the application of deep learning techniques to build the fault prediction model based on learning the semantic and syntactic features of source code, in order to improve predictive efficiency compared to traditional models.

4.1. Introduction

Deep learning models are a promising approach to learning hidden semantic features, enhancing fault prediction performance. However, sequential networks (RNN, LSTM) have difficulty fully representing syntactic and semantic information from an abstract syntax tree (AST) and capturing complex dependencies in the source code.

The Transformer architecture, based on self-attention and parallel computing capabilities, has proven effective in many areas and can overcome the problem of vanishing/exploding gradients.

This chapter proposes a Transformer-based fault prediction model, using Gensim FastText with embedding tokens from an abstract syntax tree (AST). AST is a structural representation tree of source code, which is used to describe the syntactic structure and semantic information of the source code, such as the control thread structure with the statements "while, do, for" as well as the name of the method.

Specifically, for each project file, we first build AST from the source code, and then extract the AST nodes to create token vectors. Therefore, each source code file is represented by token vectors. Next, we use a pre-trained vector word representation model, called Gensim FastText, to convert token vectors into fixed-length digital vectors. These collected vectors are then fed into the Transformer model to perform fault prediction. The main contribution of this chapter is to propose a fault prediction model based on FastText-Transformer, which is capable of learning important semantic features of software programs that greatly improve the efficiency of fault prediction. The model can capture semantic information in vector representations of source code tokens, applicable both within the same project and between projects.

In addition, we performed the Wilcoxon test to assess the degree of statistically significant difference between the FastText-Transformer model and the matching models. The results of the evaluation of 10 Java projects show that our proposed model is superior to the current new fault prediction models. The main contribution is to propose a Transformer-based fault prediction model, capable of learning the important semantic features of the software program,

significantly improve the efficiency of fault prediction for both internal projects (Within-Project Defect Prediction (WPDP)) and cross-project (Cross-Project Defect Prediction (CPDP)). The model also has a more efficient training time.

4.2. Related studies and research issues

Recent studies have applied deep learning models to extract the semantic and syntax information of the source code. Programs have a well-defined syntax that can be represented by an abstract syntax tree[173] and have been used successfully to capture source code information. Numerous studies have shown that the program's semantic information is useful for tasks such as automating code completion and detecting faults [174]. This semantic information can also be useful in describing faults in order to improve fault prediction. Specifically, this semantic information can make accurate predictions, features that need to be distinctive, that is, able to distinguish different code regions. Traditional fault prediction models often ignore the syntactic and semantic structure of the source code. Recent studies have applied deep learning to extract this information from AST.

Word Embedding is a technique for representing words/phrases as vectors in continuous space, helping machine learning algorithms understand natural language better. FastText is an efficient algorithm that treats a word as a set of n-grams of characters, allowing it to create vector representations for words that have never appeared before. This study uses Gensim FastText to learn vector representations for tokens from the AST of Java projects.

Other deep learning models such as CNN, DBN, LSTM, GRU have also been used in fault prediction. However, these sequential models are not robust enough to learn complex relationships and short contexts in source code, prone to vanishing/exploding gradients.

Research Issues:

The difference between the solution proposed in this thesis and the above studies is presented as follows. First, traditional methods use software measurements to build predictive models. However, these measurements cannot capture the syntactic and semantic features of the source code. The proposed solution can capture semantic information using FastText, a deep learning model for learning vector representations from the program's AST. Second, previous studies using sequential deep learning models such as DBN, RNN, and LSTM, which encountered the problem of disappearing gradients and burst gradients. By using a parallel Transformer architecture with self-attention and parallelization, our proposed method overcomes problems such as gradient loss and significantly shortens training time compared to sequential models such as RNN and LSTM. Overall, this method integrates three key components in a unified process to improve fault prediction accuracy. The source code is first parsed into an AST tree

to capture structural and syntactic information. Next, the tokens extracted from AST are embedded using FastText to maintain semantic similarity, including samples at the child word level. Finally, the Transformer Encoder model with its self-attention mechanism is capable of effectively capturing long-term dependencies and hierarchical relationships between source code components. This combination allows for the efficient extraction of semantic and structural features, which play a key role in identifying fault-prone snippets.

4.3. Proposed Method

The proposed method, Gensim FASTText-Transformer, consists of three main parts:

1. Source code representation and feature extraction: The javalang tool is used to convert Java files to AST. From AST, the nodes associated with the Class Node, Declaration Node, and Control Node are extracted as tokens. The Breadth-First Search (BFS) algorithm is used to browse AST and collect tokens, creating “context windows”.
2. Applying FastText to Token Embedding: The Gensim FastText model is used to convert tokens into fixed-length numerical vectors (100). FastText is capable of processing sub-word information and extra-vocabulary words, helping to create more meaningful representations for unseen tokens.
3. Transformer Encoder: Embedding vectors from FastText are fed into the Transformer architecture. Transformer uses self-attention and feed-forward mechanisms, as well as normalization and dropout layers, to process magnetic vectors. This architecture enables parallel processing, captures complex dependencies, and minimizes overfitting. The model is trained with the loss function "sparse categorical cross-entropy" and the optimizer "Adam".

Research Questions:

1. Does our FastText-Transformer outperform the state-of-the-art fault prediction approaches in WPDP?
2. Does our FastText-Transformer outperform the state-of-the-art fault prediction approaches in CPDP?

4.4. Implementation of experiments, results and analysis

The study evaluated the proposed model on ten Apache datasets (Ant, Camel, Jedit, Log4j, Lucene, Poi, Synapse, Ivy, Xalan, Xerces), which are widely used in SDP research.

Comparison models include:

- DP-Transformer
- DBN-CP
- DTL-DP
- TCA+
- And FastText-LSTM (a control model built for direct comparison with FastText-Transformer).

The performance measures are Precision, Recall, and F1-score. The Wilcoxon signed-rank test is used to assess statistical significance.

Training Parameters: Parameters such as Epochs (200), Batch-size (16), Dropout (0.2-0.25 for Transformer, 0.5 for LSTM), Learning Rate (1e-4 for Transformer, 0.001 for LSTM), Activation Function (ReLU, Softmax), and Optimizer (Adam) are adjusted to optimize performance.

4.4.1. Predicting Faults in Within-Project Defect Prediction

The results of the WPDP performance comparison are as follows:

- The FastText-Transformer model performed better than other methods across 11 of the 19 datasets.
- FastText-Transformer has an average F1-score of 0.768, Precision of 0.762, and Recall of 0.807.
- Compared to DTL-DP, Seml, DBN-CP, and FastText-LSTM, the proposed model had a higher average F1-score of 19.6%, 25.4%, 19.8%, and 7%, respectively.
- Figure 4.5 shows that FastText-Transformer outperforms DP-Transformer on Lucene and Synapse, and is equivalent on Poi.
- The statistical results (Table 4.10) show a statistically significant performance difference between FastText-Transformer and most comparison models (P-value < 0.05). A high size r effect value (>0.2) also indicates a significant difference.

This proves that FastText-Transformer is effective in capturing the semantic features of the program, improving the performance of fault prediction in the same project.

4.4.2. Predicting faults in Cross-Project Defect Prediction

The F1-score value for CPDP is as follows:

- The Gensim FastText-Transformer model performed better than the rest of the methods in terms of F1-score values across 12 of the 22 experimental sets.

- The average F1-score of Gensim FastText-Transformer in CPDP is 0.667, outperforming DTL-DP (0.618), TCA+ (0.479), DBN-CP (0.568) and FastText-LSTM (0.589) by 7.9%, 39.2%, 17.4% and 13.2% respectively.
- This superiority was also pronounced compared to FastText-LSTM on most experimental datasets (e.g., on Poi 2.5-> Synapse 1.2, FastText-Transformer reached 0.774, which is 0.274 higher than FastText-LSTM's 0.50).
- The statistical results also confirmed the statistically significant performance difference between FastText-Transformer and other models in all CPDP cases (P-value < 0.05, $r > 0.6$).

This superior performance is due to Transformer's self-attention mechanism, which improves the performance of the software fault prediction model.

4.4.3. Comparison of training time costs

Compare the training time cost between FastText-LSTM and FastText-Transformer (in seconds) for the following results:

- FastText-Transformer exhibits a distinct advantage in time performance, with significantly lower costs for all projects.
- The average training time of FastText-Transformer is 2973.26 seconds, which is significantly lower than FastText-LSTM's 4937.55 seconds.
- For example, on the Camel project, FastText-Transformer takes only 160.67 seconds compared to FastText-LSTM's 765.81 seconds.

This advantage is due to the parallel processing capabilities of the Transformer architecture, which does not depend on sequential computation like LSTM or RNN.

4.4.4. Threats to the credibility and value of the research

The study also analyzed potential threats to reliability and value.

1. External Validity: Results may not be fully generalized due to limited datasets (only Java projects from PROMISE/Apache). The future will expand to other types of applications and programming languages (Python, C++, JavaScript).
2. Internal Validity: The source code of matching studies is not available, so experimental reproduction may not be completely consistent. The use of default values for hyperparameters is also a threat, which will be improved by more thorough parameter adjustments in the future.

3. Construct Validity: Precision, Recall, F1-score are common, but other measurements may be considered for a more comprehensive assessment in the future.

4.5. Chapter summary

Chapter 4 proposed the Gensim FastText-Transformer model to improve feature learning and enhance software fault prediction performance. This model leverages Gensim FastText to convert tokens into vectors and feed them into Transformer to predict faults. Experimental results on 10 Apache projects showed that FastText-Transformer outperformed in terms of precision, recall, and F1-score on both predicting software faults within the same project and between projects. The model also outperforms many comparative models in most projects.

In the future, to increase generality, the model can be tested on datasets from various fields and programming languages, and combined with sampling techniques (oversampling, undersampling) to handle unbalanced data. In addition, it is scalable to predict faults at more granular levels such as classes, methods, or statements.

Conclusions and development directions

Conclusions

The project has succeeded in researching and proposing methods to improve the performance of software fault prediction models, especially in the context of data imbalances and the need to exploit rich semantic features. Key contributions include:

- Evaluate the effectiveness of wrapper feature selection methods: These methods, especially Recursive Feature Elimination (RFE), Equilibrium Optimizer (EO), have demonstrated the ability to reduce data redundancy and improve fault prediction performance compared to the baseline model.
- Combining advanced characterization and data sampling techniques: Research has demonstrated that the combination of RFE feature selection techniques/Autoencoders characterization and GAN variants (such as CTGAN, WGANGP, VanillaGAN) significantly improves the performance of traditional machine learning models when data is out of balance. CTGAN was identified as the most effective variant in creating new prototypes to balance the data.
- Proposed new FastText-Transformer model: This model, based on the Transformer architecture and representation from FastText, has demonstrated the ability to learn deep semantic features from the abstract syntax tree of the source code, thereby significantly improving the measurements of the evaluation of fault prediction models (Precision, Recall, F1-score) for both WPDP and CPDP. This model also shows superior efficiency in training time compared to FastText-LSTM.
- Robust empirical and statistical evaluation: Tests were conducted on popular open-source datasets and evaluated using quantitative measurements and Wilcoxon statistical testing, showing that the results obtained were highly feasible and statistically significant.

In conclusion, the project has contributed new and effective approaches in the field of software fault prediction, contributing to the goal of improving the reliability and quality of modern software systems.

Future research directions

- Extend the application to various fields and programming languages (Python, JavaScript, C++) to increase the generality of the model.
- Improve sampling strategies (oversampling, undersampling) and adjust parameters to make the model work more efficiently in complex data environments.
- Extend methods to predict faults at more granular levels such as classes, methods, or statements.

PUBLICATIONS

No.	Thể loại	Year	Publisher
1	International journal (Scopus indexed)	2026	Neural Computing and Applications, (Scopus, Q1).
2	International journal (SCIE indexed)	2025	Applied Intelligence. (SCIE, Q2, IF 3.5)
3	International journal	2025	International Journal of Electrical and Computer Engineering.
4	Domestic journal	2025	The University of Danang - Journal of Science and Technology (UD-JST)
5	Domestic journal	2024	The University of Danang - Journal of Science and Technology (UD-JST)
6	National conference proceedings	2023	National Conference on Fundamental and Applied IT Research (FAIR).
7	Proceedings of International conference (Scopus indexed, book chapter)	2023	CITA 2023: Conference on Information Technology and Its Applications. <i>Số: Part of the Lecture Notes in Networks and Systems book series.</i>
8	Proceedings of International conference	2022	ICIT 2022: Intelligence of Things: Technologies and Applications.
9	Domestic journal	2021	Journal of Research, Development and Application on Information and Communication Technology (ICT).