PhD-FSTM-2025-049
Faculty of Science, Technology and Medicine

Dissertation

Presented on the 08/04/2025 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Xueqi Dang
Born on 19th April 1998 in Shandong, China

Learning-based Test Input Prioritization for Machine Learning Systems

Dissertation Defense Committee

Dr. Yves Le Traon, Dissertation Supervisor
Professor, University of Luxembourg, Luxembourg
Director of the Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg

Dr. Mike Papadakis, Chairman
Professor, University of Luxembourg, Luxembourg

Dr. Maxime Cordy, Vice Chairman
Research Scientist, University of Luxembourg, Luxembourg

Dr. Gilles Perrouin, Member
Professor, University of Namur, Belgium

Dr. Gunel Jahangirova, Member
Professor, Kings College London, UK

Abstract

Machine learning (ML) has achieved significant success across various fields. Ensuring the reliability of ML systems through testing is essential. However, ML testing faces a major challenge: it is expensive to label each test input to assess the model's accuracy on the testing set. This is mainly due to three reasons: 1) reliance on manual labeling, 2) the large scale of test datasets, and 3) the need for domain expertise during the labeling process. Test input prioritization has become a promising strategy to mitigate the labeling cost issues, which focuses on prioritizing test inputs that are more likely to be misclassified. Enabling the earlier labelling of such bug-revealing inputs can accelerate the debugging process, therefore enhancing the efficiency of ML testing. In the existing literature, various test prioritization methods have been introduced, which can generally be classified into coverage-based, confidence-based, and mutation-based approaches. While these methods have demonstrated effectiveness in certain scenarios, they exhibit notable limitations when applied to more specialized contexts. This dissertation focuses on three specific scenarios: classical machine learning classification, long text classification, and graph neural network (GNN) classification. Specifically, for each scenario, we introduce a novel test prioritization method, as detailed in Chapters 3 to Chapter 5. Beyond proposing these new methods, we also conduct an empirical study focusing on GNN classification to explore the limitations of existing test selection approaches when applied to GNNs (cf. Chapter 6).

- **MLPrior: A New Test Prioritization Approach for Classical Machine Learning Models** To tackle the challenges in traditional ML testing, we propose a novel test prioritization method named MLPrior. MLPrior is specifically designed to leverage the unique characteristics of classical ML models (i.e., compared to DNNs, traditional ML models are generally more interpretable, and their datasets often consist of carefully engineered feature attributes) for effective test prioritization. MLPrior is built on two key principles: 1) tests that are more sensitive to mutations are more likely to be misclassified, and 2) tests that are closer to the model's decision boundary are more likely to be misclassified. Experimental results reveal that MLPrior surpasses other prioritization methods, achieving an average improvement ranging from 14.74% to 67.73%.
- **GraphPrior: A New Test Prioritization Approach for Graph Neural Networks** To enhance the efficiency of GNN testing, we propose GraphPrior, a novel test prioritization method specifically designed for GNNs. In particular, we introduce new mutation rules tailored to GNNs to generate mutated models and leverage the mutation results for effective test prioritization. The core principle is that test inputs that "kill" more mutated models are considered more likely to be misclassified. Experimental results demonstrate that GraphPrior outperforms all baseline methods, achieving an average performance improvement of 4.76% to

49.60% on natural datasets.

- **LongTest: A New Test Prioritization Approach for Long Text Files** Long texts, such as legal documents and scientific papers, present unique challenges for test prioritization due to their substantial length, complex hierarchical structures, and diverse semantic content. To address these issues, we propose LongTest, a novel approach specifically tailored for long text data. LongTest is built based on two key components: 1) a specialized embedding generation mechanism designed to extract crucial information from entire long documents, and 2) a contrastive learning framework that enhances prioritization by effectively distinguishing misclassified samples from correctly classified ones. Experimental evaluations demonstrate that LongTest outperforms baseline methods, with average improvements ranging from 14.28% to 70.86%.
- **An Empirical Study investigating the limitations of test selection approaches on GNNs** To investigate the limitations of existing DNN-oriented test selection methods in the context of GNNs, we carried out an empirical study involving 22 test selection techniques evaluated across seven graph datasets and eight GNN models. This study concentrated on three key objectives: 1) Misclassification Detection: identifying test inputs with a higher probability of being misclassified; 2) Accuracy Estimation: selecting a representative subset of tests to accurately estimate the overall accuracy of the full test set; 3) Performance Improvement: selecting retraining samples to enhance the accuracy of GNN models. Our findings indicate that the effectiveness of test selection methods for GNNs falls short when compared to their performance in the context of DNNs.

In summary, this dissertation introduces three novel test prioritization methods designed for some specific machine learning scenarios and presents an empirical study to explore the limitations of existing test selection approaches when applied to GNNs.

Always do your best. What you plant now, you will harvest later.

Og Mandino

Acknowledgements

I would like to extend my sincere appreciation to all those who have contributed to my academic journey, offering their expertise, advice, and motivation.

Firstly, I would like to express my sincere gratitude to my supervisor, Prof. Yves Le Traon, for his continuous support, trust, and encouragement throughout my PhD studies. His insightful feedback, patience, and academic rigor have profoundly influenced my research and personal growth. His dedication to excellence has been a source of inspiration, motivating me to pursue my work with determination.

I also deeply appreciate my daily supervisor, Prof. Michail Papadakis, for his invaluable guidance. His expertise and willingness to engage in detailed technical discussions significantly enhanced my research contributions. Additionally, I extend my heartfelt thanks to my co-supervisor, Dr. Maxime Cordy, for his mentorship and encouragement.

I would like to express my appreciation to Dr. Yinghua Li, whose collaboration, discussions, and assistance were instrumental throughout my PhD. Furthermore, I am grateful to all my co-authors for their contributions, collaboration, and constructive feedback. Their expertise and dedication have significantly enriched my research experience.

I would also like to extend my heartfelt thanks to the members of my PhD defense committee, including Prof. Yves Le Traon, Prof. Michail Papadakis, Dr. Maxime Cordy, Prof. Gilles Perrouin, and Prof. Gunel Jahangirova. It is a great honor to have them evaluate my dissertation, and I deeply appreciate their time and thoughtful reviews, which have helped refine my work.

I am also grateful to my colleagues and friends from the Serval (SnT) research group for their insightful discussions and valuable suggestions and for fostering a collaborative research environment. Their companionship and shared experiences made my PhD journey more fulfilling.

I would like to acknowledge the support of the Luxembourg National Research Fund (FNR) under the AFR PhD grant 17036341.

Finally, my deepest appreciation goes to my parents for their unconditional support, encouragement, and belief in me. Their unwavering confidence in my abilities has been a constant source of strength, enabling me to pursue my academic aspirations with determination.

Xueqi Dang
University of Luxembourg
April 2025

Contents

List of Figures

List of Tables

1 Introduction

In this chapter, we begin by introducing the motivation for Learning-based Test Input Prioritization in Machine Learning Systems. We then discuss the limitations of existing machine learning test prioritization methods within the specific contexts of our focus. Finally, we outline the contributions of this dissertation and provide a roadmap for the chapters that follow.

Contents

1.1 Motivation

Machine learning (ML) has revolutionized numerous fields, such as image recognition, natural language processing, and recommendation systems. However, ensuring the reliability and accuracy of ML models remains a significant challenge. Testing is one of the most widely used methods to ensure the quality of ML systems. Nevertheless, a major challenge in ML testing lies in the high cost of labeling test data. This issue arises primarily from three factors: 1) manual labeling remains the mainstream approach, typically requiring multiple annotators to maintain accuracy and consistency; 2) test datasets can be large-scale; and 3) labeling can demand domain-specific expertise from professionals in the relevant field, further increasing the cost. For example, when using the traditional ML model XGBoost to detect chronic kidney disease (CKD) [1], labeling the CKD dataset for model training and testing requires specialized medical knowledge to accurately determine whether a patient has CKD.

To address the high labeling cost issue, test input prioritization has emerged as a promising solution. This approach focuses on identifying and prioritizing test inputs that are more likely to be misclassified by the ML model. These inputs are also referred to as bug-revealing inputs. Early identification and labeling of these bug-revealing inputs can accelerate the debugging process, thereby enhancing the overall efficiency of ML testing. In the literature, numerous test prioritization methods [2, 3] have been proposed. These methods are broadly categorized into three main types: 1) Coverage-based approaches [4, 5, 6]; 2) Confidence-based approaches [3, 7], and 3) Mutation-based approaches [2]. Coverage-based approaches prioritize test inputs by analyzing the neuron coverage achieved by the DNN model. Confidence-based methods, on the other hand, focus on identifying potentially misclassified test inputs by measuring the model's output confidence. A notable confidence-based method is DeepGini [3], which utilizes the Gini score to quantify the model's confidence for each test. Tests where the model shows lower confidence are considered more likely to be misclassified and thus will be prioritized higher. Mutation-based approaches focus on designing new mutation operations and analyzing the mutation results for each test input to guide test prioritization.

However, although existing prioritization approaches demonstrate effectiveness in certain cases, they still face limitations in specific scenarios. In the following, we provide a detailed explanation of the limitations of the existing prioritization methods in the contexts of classical machine learning classification, Graph Neural Networks (GNNs) classification, and long-text classification, respectively. These limitations serve as the core motivations for proposing new methods in these areas.

1.2 Limitations of Existing Methods

1.2.1 Classical Machine Learning classification

Machine learning can be generally divided into two main categories: classical machine learning and deep learning. Classical machine learning algorithms, including methods like XGBoost [8] and decision trees [9], typically demonstrate better interpretability when compared to deep neural networks (DNNs). Interpretability in this context refers to the degree to which the internal mechanisms and decision-making processes of a model are comprehensible and explainable to humans.

When applying existing test prioritization approaches to the context of traditional machine learning fields, the following limitations arise:

- **Single Dimension on Binary Classification Models** A Binary classification model categorizes a given test input into two classes. If the model predicts that the test input has a probability p of belonging to the first class, then it believes that there is a probability of $1^{\smile}p$ that the test belongs to the second class. In this case, the closer p is to 0.5, the more uncertain the model is about this prediction. Therefore, tests with p values closer to 0.5 will be consistently prioritized higher regardless of the specific confidence-based test prioritization method used. Therefore, all confidence-based methods will yield the same test prioritization results.
- **Absence of Model-specific Insights**: Confidence-based prioritization methods treat the model as a black box, relying solely on the prediction probability vector generated by the model for each test to perform test prioritization. However, classical machine learning models are often white-box models, meaning that their internal information can also be utilized for test prioritization. Confidence-based methods fail to leverage this interpretability and transparency characteristic of classical machine learning models, ignoring the valuable internal information these models provide for test prioritization.
- **Ignorance of Attribute-level Features** Traditional ML models typically use tabular data as inputs, which differs from the input format used by DNN models. The attribute features of test inputs provide a direct representation of tests within the feature space and reveal their proximity to the model's decision boundary, thereby contributing to test prioritization. However, confidence-based approaches fail to leverage this critical feature information for test prioritization.

1.2.2 GNN classification

Unlike traditional neural networks that operate on fixed-sized vectors, GNNs are specifically designed to process graph-structured data. In this context, a graph typically refers to a data structure comprising two components: nodes (vertices) and edges. Directly applying existing DNN-oriented test prioritization approaches to GNNs presents the following limitations:

- Confidence-based techniques fail to account for the interdependencies between test inputs in GNNs, which are critical for GNN inference. These prioritization approaches typically treat test sets as collections of independent samples with no connections. However, GNN test inputs are represented as graph-structured data, where nodes are interconnected by edges.
- The effectiveness of uncertainty-based test prioritization approaches can be affected when facing some specific adversarial attacks. For instance, if an attack is designed to generate test inputs that increase the model's predicted probability for incorrect classification, the effectiveness of uncertainty metrics can be impacted. This is because uncertainty-based methods assume that test samples where the model exhibits high uncertainty are more likely to be misclassified and, therefore, prioritize these samples. However, in such cases, many test samples that the model is confident about can actually be misclassified, which reduces the effectiveness of confidence-based prioritization approaches.
- Existing research [3] has shown that coverage-based methods are both less effective and less efficient than confidence-based test prioritization approaches.

1.2.3 Long Text Classification

Text classification focuses on categorizing text documents into predefined labels. Compared to short texts, such as social media posts, emails, or product reviews, long text classification (e.g., legal documents, academic papers, and technical reports) poses unique challenges. These challenges arise from their extended length, complex structures, and richer semantic content. When applying existing DNN test prioritization approaches to long text classification, the following limitations arise:

- Confidence-based test prioritization approaches rely solely on the probability prediction vector of each test input from the model's final layer and use these vectors to measure the model's prediction confidence for each test. These methods ignore the rich semantic and hierarchical information inherent in the test inputs (long text), which limits their ability to effectively prioritize tests.
- Mutation-based test prioritization methods are generally designed for short texts, where the proposed mutation operators typically introduce small changes (e.g., modifying a few characters within a word). However, long texts contain extensive content and a larger number of words, making these minor mutations less effective. Consequently, mutation-based methods are not well-suited for test prioritization in long-text classification scenarios.
- From the existing work [3], the coverage-based test prioritization methods have been shown to be less effective and more computationally expensive than confidence-based prioritization approaches.

1.3 Contributions

Due to the limitations of existing test prioritization methods when applied to the aforementioned special scenarios, we propose a novel test prioritization method tailored specifically to each scenario. Moreover, specifically for the GNN classification scenario, we conduct an empirical study to thoroughly examine the limitations of existing test prioritization methods in this context. Therefore, the main contributions of this thesis are as follows:

- **We propose MLPrior, a novel test prioritization approach for classical machine learning (ML) models.** MLPrior leverages the unique characteristics of classical ML models to perform test prioritization. Compared to deep neural networks (DNNs), classical ML models are generally more interpretable, and their datasets typically consist of carefully engineered feature attributes. Based on these characteristics, the working mechanism of MLPrior relies on two core ideas: 1) prioritizing tests that are sensitive to mutations, and 2) prioritizing tests that are near the decision boundary. Experimental results demonstrate that MLPrior outperforms the compared existing methods, achieving improvements ranging from 14.74% to 67.73%.

  This work has been accepted by the IEEE Transactions on Software Engineering (TSE) in 2024.

- **We propose GraphPrior: a novel test prioritization approach for graph neural networks (GNNs).** GraphPrior leverages the unique characteristics of GNNs and graph-structured datasets for test prioritization. Unlike DNNs, where test inputs are independent of one another, GNN test inputs are typically

represented as graphs with complex interdependencies. Building on this, GraphPrior introduces newly proposed mutation rules specific to GNNs to prioritize test inputs, based on the principle that tests that "kill" many mutated models are more likely to be misclassified. GraphPrior outperforms all the compared test prioritization approaches.

This work has been accepted by the ACM Transactions on Software Engineering and Methodology (TOSEM) in 2023.

- **We propose LongTest, a novel test prioritization approach for long text classification.** Compared to short texts such as social media posts, long texts (e.g., scientific papers and legal documents) pose unique challenges for test prioritization due to their substantial length and diverse semantic content. To address these challenges, our proposed LongTest incorporates two core components specifically designed for long texts: a specialized embedding generation mechanism for extracting text representations from lengthy files, and a contrastive learning framework to effectively differentiate between misclassified and correctly classified inputs. LongTest outperforms all the compared test prioritization approaches, achieving average improvements ranging from 14.28% to 70.86%.

  This work is currently under review in ACM Transactions on Software Engineering and Methodology(TOSEM) in 2025.

- **We conducted an empirical Study to explore the limitations of current test selection approaches on GNNs.** In the empirical study, we investigate the limitations of existing test selection methods when applied to GNNs. Our investigation focuses on three critical aspects: test selection for misclassification detection, test selection for accuracy estimation, and test selection for GNN model retraining. The study evaluates 22 test selection approaches across 7 graph datasets and 8 GNN models. The results demonstrate that DNN test prioritization methods do not achieve the same level of effectiveness when applied to GNNs as they do in the context of DNNs.

  This work has been accepted by Empirical Software Engineering (EMSE) in 2024.

1.4 Roadmap

The roadmap of the dissertation is illustrated in Figure 1.1. Chapter 2 provides the **Background** for the thesis, covering topics including test optimization in DNN Testing, mutation testing, and contrastive learning. Chapter 3 introduces our proposed test prioritization approach, **MLPrior**, specifically designed for classical machine learning models. Chapter 4 details our proposed test prioritization approach, **GraphPrior**, tailored for Graph Neural Networks. Chapter 5 describes our proposed test prioritization approach, **LongTest**, designed for long text files. Chapter 6 presents our **empirical study**, investigating the limitations of current DNN test prioritization approaches when applied in the context of Graph Neural Networks. Chapter 7 reviews **related work**, including deep neural network testing, test selection

for DNNs, and test optimization for traditional software. Finally, Chapter 8 concludes the dissertation and discusses directions for future research.

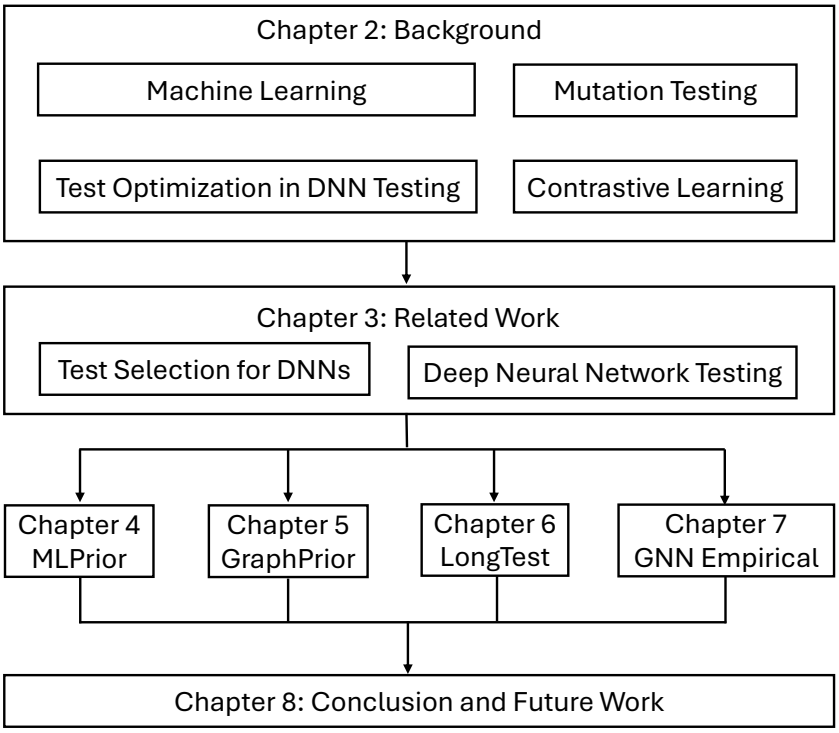


Figure 1.1: Roadmap of this dissertation

2 Background

This chapter introduces the essential background to establish a foundation for understanding this dissertation. It first provides a description of machine learning, with a particular focus on the areas where we propose new test prioritization approaches or conduct empirical studies on existing prioritization approaches. Then, it presents the relevant background on Test Optimization in DNN Testing, Mutation Testing, and Contrastive Learning.

Contents

2.1 Machine Learning

Machine Learning (ML), a branch of artificial intelligence, enables systems to make decisions or predictions based on data. Its widespread adoption in various fields demonstrates its significant utility, especially in safety-critical areas such as finance [10], insurance [11], and healthcare [12]. In our work, we focus on proposing new test prioritization approaches and conducting empirical studies in three specific machine learning contexts: classical machine learning, deep neural networks, and graph neural networks, as existing DNN-oriented test prioritization approaches exhibit certain limitations. In the following, we provide some basic background knowledge for these fields.

2.1.1 Classical Machine Learning

Broadly, ML techniques can be categorized into classical machine learning and deep learning [13]. Classical machine learning includes methods such as decision trees [9], logistic regression [14], and naive bayes [15], which typically rely on structured data and predefined algorithms to make predictions. In contrast, deep learning utilizes multiple layers of nonlinear processing units to extract and transform features [16]. Deep learning techniques include convolutional neural networks (CNNs) [17] for image analysis, recurrent neural networks (RNNs) [18] for sequential data, and transformers for natural language processing.

In contrast to deep learning, classical machine learning algorithms provide better interpretability. Interpretability refers to the degree to which the decision-making processes of a model can be understood by humans. This attribute is particularly crucial in fields such as healthcare [19] and finance [20], where understanding the rationale behind predictions is as important as the predictions themselves, due to the potential ethical, legal, and safety implications in these domains. As a result, classical machine learning models retain unique advantages in certain application areas.

2.1.2 Deep Neural Networks

Deep Neural Networks (DNNs) consist of several hierarchical layers, each containing interconnected processing units referred to as neurons [21, 22]. These neurons are linked by weighted connections. During training, these weights are iteratively refined through optimization algorithms, such as gradient descent [23], to minimize a predefined loss function that quantifies the difference between the model's predictions and the actual target values. The training process relies heavily on the input data, which guides the adjustment of these weights.

One typical application of DNNs is text classification [24], which focuses on assigning textual data to specific categories based on its content. For example, in sentiment analysis [25], DNNs can be used to identify the emotional tone of texts, such as determining whether a review expresses a positive, negative, or neutral sentiment. In spam filtering [26, 27], DNNs can help distinguish malicious messages from legitimate communications, enhancing email security. Specifically, in the field of text classification, the target text can be either short or long. Short texts, such as social media posts or product reviews, are typically brief and straightforward. Conversely, long texts, such as legal documents or scientific articles, present unique challenges due to their extended length and complex hierarchical structures [28].

In one of our works (cf. Chapter 5), we focused on proposing new test prioritization

approaches, specifically in the field of long text classification. By leveraging the unique characteristics of long texts, we designed an embedding generation mechanism tailored for long text files and employed contrastive learning to better distinguish correctly classified and misclassified test inputs. These two components facilitated more effective test prioritization. Experimental evaluations demonstrated that our test prioritization method consistently outperformed all baseline approaches.

2.1.3 Graph Neural Networks

Unlike traditional deep neural networks (DNNs) that operate on fixed-sized vectors, Graph Neural Networks (GNNs) are specifically designed to process graph-structured data, which can vary in size and structure. Graph-structured data typically refers to data that is represented as nodes (entities) and edges (relationships) within a graph. Examples include social networks, where individuals are nodes connected by edges representing relationships; molecular structures, where atoms are nodes and bonds are edges; or traffic networks, where intersections are nodes and roads are edges. Compared to DNNs, GNNs are more effective at modeling the complex dependencies and relationships inherent in such data.

The classification tasks of GNNs mainly include three types: node classification, edge classification, and graph classification. Node classification [29] focuses on assigning labels to individual nodes within a graph. For example, in citation networks, node classification can be used to predict the research field of a specific paper based on its connections with other papers, as shown in the widely studied Cora and PubMed datasets [30]. Edge classification focuses on classifying the edges within a graph, where edges represent relationships between nodes. For instance, in the field of biological networks, GNNs can be employed to predict the binding affinity between a protein and a small molecule, with the binding affinity being modeled as an edge in the graph. Graph classification, on the other hand, focuses on classifying entire graphs, such as predicting molecular properties based on chemical structures [31].

2.2 Test Optimization in DNN Testing

To improve the efficiency of DNN testing, test optimization [3, 32, 33, 7, 2, 34] has emerged as a crucial area of focus. There are mainly two categories of approaches to optimize the DNN testing process, which are test selection [35, 36] and test prioritization [3, 7, 2]. Test selection aims to select a subset of test cases from the entire test set to estimate the accuracy of the original set. Li *et al.*[36] proposed a test selection method called Cross Entropy-based Sampling (CES), which works by minimizing the cross-entropy between the selected and original test sets, thereby ensuring that the selected subset maintains a similar distribution to the complete test set. Chen *et al.* [35] proposed a clustering-based test selection approach, called PACE, which first clusters all test cases and then employs the MMD-critic algorithm [37] to select prototypes from each cluster. For test inputs not assigned to any cluster, PACE applies adaptive random testing to perform selection.

Test prioritization, on the other hand, aims to change the order of test execution so that tests that are more likely to be misclassified are prioritized higher. Unlike test selection, in the process of test prioritization, no test inputs are discarded. By prioritizing potentially misclassified test inputs, it ensures that developers focus their limited labeling resources on bug-revealing test cases, thereby enhancing debugging efficiency. Feng *et al.* proposed DeepGini [3], a well-known test prioritization method

leveraging model confidence. This approach calculates a Gini score for each test case to estimate the model's confidence and ranks the test cases based on these scores. Weiss *et al.* [7] carried out a comprehensive evaluation of various test prioritization techniques, demonstrating that some relatively simple uncertainty-based methods, such as Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy, perform effectively. Wang *et al.* [2] proposed a mutation-based test prioritization approach called PRIMA. PRIMA utilized newly introduced mutation operators to generate mutation results for each test input, thereby achieving test prioritization.

2.3 Mutation Testing

Mutation testing [38, 39] involves introducing minor, deliberate modifications to program code, referred to as mutations, to assess the effectiveness of test cases. These mutations simulate potential faults that may occur during software development. A high-quality test suite is expected to detect the introduced mutants [40]. In this context, the term "kill" is commonly used to describe whether a test case successfully detects a mutation in the code. A mutant is considered "killed" if executing a test case reveals a behavioral difference between the original program and its mutated version, indicating that the test case has identified the mutation. Conversely, if the test case cannot distinguish between the behavior of the original program and the mutated version, the mutant is considered "survived", suggesting that the test case fails to capture certain defects. The proportion of mutants killed by test cases serves as a metric to evaluate the effectiveness and coverage of the test cases. A higher kill rate generally indicates that the test cases can effectively detect faults. Mutation testing has gained significant attention in both academic research and industry [41, 42, 43, 44].

2.4 Contrastive Learning

Contrastive Learning [45] is a self-supervised learning method that learns effective data representations by pulling similar samples closer together in the representation space and pushing dissimilar samples further apart. The core idea of contrastive learning is to leverage the relative relationships between samples without requiring explicit labels, typically through the construction of positive and negative sample pairs. In this context, positive sample pairs refer to pairs of samples from the same category whose representation vectors in the embedding space are supposed to be close. Negative sample pairs, on the other hand, are pairs of samples from different categories whose representation vectors in the embedding space are supposed to be as far apart. The objective of contrastive learning is to optimize a contrastive loss function (e.g., InfoNCE [45]), therefore maximizing the similarity of positive sample pairs while minimizing the similarity of negative sample pairs.

3 Related Work

This chapter introduces the related work of this dissertation, including Test Selection for DNNs and Deep Neural Network Testing.

Contents

3.1 Test Selection for DNNs

Evaluating deep learning models presents significant challenges due to the high labelling cost for annotating test cases. Test selection methods aim to address this issue by selecting and labeling a subset of test data, thereby reducing the overall effort required for labeling. Recent studies have concentrated on two main areas: misclassification detection and accuracy estimation.

Misclassification detection techniques focus on prioritizing test inputs that are more likely to be misclassified. These inputs are valuable for quickly debugging DNN systems and retraining DNN models to enhance their performance. In the literature, Feng *et al.* introduced DeepGini, which utilizes confidence-based metrics to rank tests, with tests exhibiting higher uncertainty (measured by Gini scores prioritized higher. Weiss *et al.* [7] evaluated a large set of test input prioritization strategies for DNNs, particularly emphasizing uncertainty-based metrics like Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy. These metrics have proven effective in identifying potentially misclassified test inputs. Ma *et al.* [46] proposed selecting test inputs based on uncertainty metrics, such as the maximum probability score and variance score, demonstrating that uncertainty can guide the identification of informative inputs. Additionally, Hu *et al.* [47] investigated the limitations of existing test selection methods for active learning, which uses selected tests to retrain DNNs. Their findings revealed that different test selection approaches lead to models of varying quality.

Accuracy estimation approaches, on the other hand, aim to identify representative test cases that can approximate the accuracy of the entire dataset. Li *et al.* [36] introduced CES, which minimizes the cross-entropy between the selected and full test sets, ensuring that the selected subset of tests maintains the original distribution. Chen *et al.* [35] developed PACE, a technique that selects representative test inputs through clustering, prototype selection, and adaptive random testing. PACE first groups test inputs based on their testing capabilities, then applies the MMD-critic algorithm [37] to identify prototypes within each cluster.

3.2 Deep Neural Network Testing

Beyond test selection, other areas, such as evaluating the adequacy of deep neural networks [4, 5, 48, 49, 50], have also attracted significant attention. Pei *et al.* [4] proposed the concept of neuron coverage as a measure to evaluate how thoroughly a test set explores the decision logic of a DNN model. Ma *et al.* [5] introduced DeepGauge, a framework that incorporates multiple coverage metrics to assess the adequacy of test sets for DNNs. DeepGauge tracks neuron activation patterns to evaluate how inputs traverse the decision-making logic of a DNN to evaluate the quality of test cases. Kim *et al.* [49] proposed surprise adequacy as a novel technique for evaluating the quality of test inputs in datasets. This method measures the degree of "surprise" in test inputs compared to the training data, with surprise defined based on variations in neuron activation patterns triggered by the inputs.

Dola *et al.* [51] proposed the IDC framework aimed at evaluating the adequacy of tests for DNNs in a black-box context. This framework leverages a Variational Autoencoder (VAE) to transform test inputs into feature vectors, creating a defined coverage space. Coverage in this space is then measured using metrics from Combinatorial Interaction Testing (CIT). Riccio *et al.* [52] introduced the notion

of "mutation adequacy" to measure the quality of test sets in identifying mutations in DNN models. To achieve this, they developed DEEPMETIS, a tool specifically designed to enhance the mutation adequacy of test sets, thereby improving their fault detection performance.

This thesis includes four main contributions. The first three projects focus on proposing new test prioritization methods to enhance the efficiency of DNN testing. We focused on three special scenarios: classical machine learning model classification, graph neural network classification, and long-text classification. The final project conducts an empirical study to explore the limitations of existing test selection methods in the context of graph neural networks.

4 Test input prioritization for Machine Learning Classifiers

In this chapter, we propose a novel test prioritization approach called MLPrior, which focuses on prioritizing test inputs that are more likely to be misclassified by classical machine learning (ML) classifiers. MLPrior addresses the limitations of existing test prioritization approaches: 1) Coverage-based methods are inefficient and time-consuming; 2) Mutation-based methods cannot be adapted to classical ML models due to incompatible model mutation operators; and 3) Confidence-based methods are restricted to a single dimension when applied to binary ML classifiers, relying solely on the model's prediction probability for one class. MLPrior leverages two key premises for test prioritization: 1) test inputs that are more sensitive to mutations are more likely to be misclassified, and 2) test inputs closer to the model's decision boundary are more likely to be misclassified. By prioritizing potentially misclassified test inputs, testers can allocate labeling resources more effectively, enhancing debugging efficiency.

This chapter is based on the work published in the following research paper:

- **Xueqi Dang**, Yinghua Li, Mike Papadakis, Jacques Klein, Tegawendé F. Bissyandé, Yves Le Traon. Test input prioritization for Machine Learning Classifiers. IEEE Transactions on Software Engineering (TSE). Accepted for publication on Dec. 25, 2023.

Contents

4.1 Introduction

Machine learning classifiers have seen remarkable success in various domains [53], including image recognition [54], natural language processing [55, 56], and recommendation systems [57, 58]. However, the prevalence of black-box models, especially in deep learning, has raised concerns about their lack of interpretability, which refers to the extent to which a model's internal mechanism and decision-making processes can be comprehended and explained transparently to humans. Interpretability becomes particularly vital in safety-critical domains like healthcare and finance [59], where model decisions can profoundly impact individuals' lives and societal well-being.

Compared to black-box models, classical machine learning (ML) algorithms (e.g., XGBoost [8], decision tree [9] and logistic regression [14]) offer more interpretable solutions, making them an appealing choice for domains that prioritize transparency and comprehensibility.

While classical ML classifiers are inherently interpretable, ensuring their accuracy and reliability remains a challenge. Testing is a fundamental practice for ensuring the quality of ML systems. However, a significant challenge in ML testing is the labeling cost issue [13] (i.e., labeling test inputs to verify the correctness of predictions can be costly). This challenge arises due to several factors: 1) manual annotation is still the mainstream for labeling; 2) test sets can be large-scale, which increases labeling efforts; 3) domain-specific knowledge can be required in certain domains for labeling tabular data, such as the medical domain [1, 60, 61]. For instance, when applying XGBoost for chronic kidney disease (CKD) detection [1], labelling the CKD dataset for model training/testing requires specialized medical expertise to determine whether a patient has CKD.

To deal with the labelling cost problem, one intuitive solution is to prioritize tests that can cause the ML model to behave incorrectly (i.e., inputs that are more likely to be misclassified by the model). Early identification and labelling of such tests can save the manual labelling effort and enhance the overall efficiency of the testing process. In the literature, various test prioritization approaches [2, 3] have been proposed in the field of DNN testing. These techniques can be broadly classified into three categories: coverage-based [4, 5, 6], confidence-based [3, 7] and mutation-based [2] approaches.

Coverage-based approaches prioritize test inputs based on the neuron coverage of DNNs. Confidence-based methods identify possibly-misclassified test inputs by quantifying the classifier's output confidence for each test. One notable confidence-based approach is DeepGini [3], which leverages the Gini score as a metric to quantify confidence levels for effective test prioritization. Recently, Weiss *et al.* [7] conducted a comprehensive study to assess existing test prioritization methods, containing the evaluation of a series of confidence-based metrics, including Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy. Mutation-based techniques propose a set of mutation operations and utilize the mutated results for test prioritization. While these approaches have made considerable progress in prioritizing potentially-misclassified test inputs, they still face certain challenges and limitations.

First, prior studies [3] have demonstrated that coverage-based methods are ineffective and time-costly compared to confidence-based approaches. Second, the mutation-based test prioritization approach, PRIMA [2], is not applicable to classical ML models due to the lack of adapted model mutation operators. Third, while

confidence-based test prioritization approaches can be adapted for classical ML models, there are several limitations associated with their application in this context. We outline the main limitations as follows. Specific details can be found in the background section (cf. Section 7.2).

- **Single dimension on binary classification models** Binary classification models categorize test inputs into two classes, and in confidence-based approaches, the likelihood of a test being misclassified primarily relies on the model's prediction probability p. Tests with p values closer to 0.5 will be consistently prioritized regardless of the specific method used, as demonstrated through experimental results.
- **Lack of model-specific insights** Confidence-based approaches, viewing the model as a black box and relying solely on its prediction probabilities, do not take into account the transparency and interpretability provided by classical ML models, leading to suboptimal prioritization.
- **Ignoring attribute features** Confidence-based methods neglect the attribute features of classical ML test datasets, which can directly map tests into space and indirectly reflect the distance between samples and the model's decision boundary. However, confidence-based approaches ignore this crucial feature information in the process of test prioritization.

In this paper, we propose MLPrior (Classical **ML**-oriented Test **Prior**itization), a test prioritization approach specifically tailored for classical machine learning (ML) models. MLPrior addresses the aforementioned limitations, leveraging the characteristics of classical ML classifiers (i.e., interpretable models and carefully engineered attribute features) to prioritize test inputs. The core ideas behind MLPrior are twofold: 1) tests more sensitive to the injected mutations are more likely to reveal bugs, and 2) test inputs closer to the decision boundary of the model are more likely to be predicted incorrectly. Both premises have been validated by existing studies [62, 63, 64, 48], with a detailed explanation provided in the Background section. Building upon the aforementioned premise, MLPrior utilizes the characteristics of classical ML classifiers to prioritize test inputs, addressing the limitations of confidence-based methods in the following way.

- **Premise 1 - tests more sensitive to the injected mutations are more likely to reveal bugs** Based on this premise, we design mutation rules specifically based on the characteristics of classical ML models and their datasets.
  1) Model mutations. Leveraging the white-box nature of most classical ML models, we design mutation rules specifically tailored for classical ML models. These rules involve modifying the model's architecture parameters or weight parameters to perform model mutations.
  2) Input mutations. Considering the tabular format of classical ML datasets, which is different from the complex data structures of DNN datasets (r.g., text and images), we design input mutation rules specifically tailored for classical ML datasets.
- **Premise 2 - test inputs closer to the decision boundary of the model are more likely to be predicted incorrectly.** To effectively capture the spatial relationship between a test input and the decision boundary, we aim to transform the attribute features of each test into a vector to indirectly reveal the underlying proximity between the input and the decision boundary. Recognizing the carefully-selected features of the classical ML test set, we design transformation

rules to convert the original attributes of each test into a feature vector for test prioritization.

Using model mutation rules and input mutation rules, we create a feature vector for each test. More specifically, we generate mutants based on the mutation rules. These mutants are then executed to generate mutation features for the purpose of assessing the sensitivity to the injected mutations. As a result, we obtain three types of features for each test: model mutation features (MMF), input mutation features (IMF), and original attribute features (OAF).

- **Model mutation features (MMF)** MMF can capture the impact of model mutations on a test input. Here, if an input can kill many mutated models (i.e., the predictions for this input via the mutated models and the original model are different), indicating that this input is sensitive to model mutations, MLPrior considers this input more likely to be misclassified.
- **Input mutation features (IMF)** IMF can capture the impact of mutations on test inputs. If the prediction result for a given test input is different from that of many of its mutated inputs, indicating that the predictions for the input are sensitive to the mutations, MLPrior considers this input more likely to be misclassified.
- **Original attribute features (OAF)** OAF can capture the spatial relationship between a test input and the decision boundary. It directly reflects the original attribute information of each test.

MLPrior combines three types of features for each test input in the target test set to generate a final feature vector. This vector is then used by a pre-trained ranking model to effectively predict the probability of misclassification for that input. MLPrior offers several advantages:

- **Generality**: MLPrior can be adapted to a wide range of classical ML models by making simple adjustments to the model mutation rules (i.e., enabling them to target the architecture parameters or weight parameters of the evaluated model).
- **Efficient**: The total duration for test prioritization using MLPrior is around 20 seconds, involving model/input mutation, feature generation, ranking model training, and test prioritization. One crucial factor is that MLPrior does not require any retraining operations in the model mutation process. Mutations are generated by directly modifying the architecture parameters or weight parameters of the evaluated models.
- **Model-specific insights** Compared to confidence-based test prioritization approaches, MLPrior leverages the interpretability characteristic of classical ML models and introduces mutations through modification of the model's architecture parameters or weight parameters, thus achieving effective test prioritization.
- **Attribute feature inclusion** In contrast to DNN test data, classical ML datasets typically possess lower-dimensional features, rendering them more cost-effective and time-efficient for test prioritization. Moreover, these features are typically carefully selected by domain experts, providing a direct reflection of attribute information for each test input. Our proposed approach MLPrior is designed to leverage the attribute features of ML test sets for test prioritization.

MLPrior demonstrates broad applicability across various contexts. One specific application pertains to banking loan operations, where classical ML models are employed to determine whether a loan can be granted to a user. In this particular scenario, classical ML models utilize a set of user attributes (e.g., gender, age, and

transaction history) to predict the viability of granting a loan to a user. Incorrect predictions can lead to significant losses for the bank. For instance, if the bank mistakenly grants a loan to a user without the ability to repay, these users can fail to meet their repayment obligations, increasing the risk of default and causing damage to the bank's assets. In this context, MLPrior can identify and prioritize users who are more likely to be misclassified by the model. Consequently, two main advantages arise: Firstly, these potentially misclassified users can be prioritized for manual inspection, resulting in a decrease in losses caused by inaccurate predictions generated by the model. Secondly, developers can manually inspect the attributes of misclassified users and analyze which attributes led to prediction errors, using this information to optimize the model.

We conducted an extensive study to evaluate MLPrior's performance utilizing 185 subjects (i.e., paired datasets and ML models). The evaluation encompassed different types of test inputs, including natural data, mixed noisy data, and fairness data. Ensuring fairness in machine learning is essential to prevent bias and discrimination against specific groups during predictions. Fairness has become a critical ethical consideration in diverse machine learning domains, including recruitment, loan approvals, and medical diagnosis [65]. In these domains, the absence of fairness can lead to unjust treatment of particular groups, affecting individuals' lives and rights. Therefore, the evaluation of MLPrior's effectiveness on fairness datasets assumes crucial importance. To generate the fairness datasets, we followed the approach of prior research [66, 67]. Specifically, we selected a group of test inputs and modified their gender and age attribute values while retaining their original labels. Moreover, we carefully selected a group of test prioritization approaches that can be adapted to prioritize test inputs in the context of classical ML models as the comparative methods, which have been demonstrated effective in existing studies [7, 3]. Additionally, we utilize random selection as the baseline approach.

The experimental results demonstrate the superior performance of MLPrior compared to existing methods, with an average improvement of 14.74%∼66.93% on natural datasets, 18.55%∼67.73% on mixed noisy datasets, and 15.34%∼62.72% on fairness datasets. We publish our dataset, results, and tools to the community on Zenodo [1].

To sum up, our work has the following major contributions:

- **Approach.** We propose MLPrior, a novel test prioritization approach specifically designed for classical ML models.
- **Study.** We conduct an extensive study based on 185 subjects involving natural, mixed noisy, and fairness test inputs. We compare MLPrior with existing DNN test prioritization approaches. Our experimental results demonstrate the effectiveness of MLPrior.
- **Performance Analysis.** We assess the influence of various ranking models on MLPrior's effectiveness. Furthermore, we evaluate the contributions of different types of features to MLPrior's effectiveness. Additionally, we explore the impact of parameter settings on MLPrior's effectiveness.

[1] https://zenodo.org/records/10150392

4.2 Background

4.2.1 Machine Learning and ML testing

Machine Learning (ML) has gained widespread adoption in various domains, demonstrating significant utility in safety-critical sectors like autonomous vehicle systems [68] and medical intervention protocols [12]. Existing literature [13] pointed out that ML can be broadly classified into two primary branches: classical Machine Learning [69, 8] and Deep Learning [70, 71]. Classical Machine Learning encompasses a range of approaches, including decision trees [9] and logistic regression [14]. These classical algorithms remain widely employed in various industrial applications [72, 37]. DNNs consist of interconnected nodes (neurons) organized in layers, with each layer responsible for learning and abstracting different levels of features from input data. In contrast to DNNs, classical ML models are generally more interpretable [73]. Interpretability in machine learning refers to the degree to which a model's internal mechanisms and decision-making processes can be understood and transparently explained to humans. Interpretability is crucial in domains where transparency and interpretability are essential, such as healthcare [19] and finance [20]. Therefore, classical machine learning models retain distinct advantages in certain application domains.

In order to emphasize the importance of interpretability in safety-critical domains, we present several typical harms caused by black-box ML systems in the financial and healthcare industries:

1) Risk Management Challenges in Finance Weber *et al.* [74] highlighted that, in the financial field, a high degree of transparency and interpretability is required for effective risk management. The lack of interpretability in black-box models can make it challenging for financial institutions to understand how decisions are made, thereby increasing the difficulty of risk management.

2) Legal and Ethical Issues in Finance Chen *et al.* [75] pointed out that, according to legal and ethical principles, financial companies are required to provide clear explanations for the reasons behind specific loan application rejections. However, with black-box models, loan applicants are unaware of how their scores are calculated. Even if model explanations are provided, there can be a disconnect between the explanations for loan rejection and the actual model calculations, as the explanations could be created after the fact.

3) Trust Issues in Healthcare Adadi *et al.* [76] discussed the constrained acceptance of black-box models in clinical settings due to trust and transparency issues. Moreover, Verdicchio *et al.* [77] raised a vital question: "If doctors cannot understand why a black-box model diagnoses, why should patients trust the treatment recommendations?". This implies that black-box models lack interpretability, making it difficult to explain the fundamental reasons behind their diagnostic or treatment recommendations. Therefore, patients and doctors can be skeptical of the system's suggestions and even refuse to follow its recommendations because they cannot be certain if these recommendations are based on sound medical reasoning. This lack of trust and understanding can significantly affect patients' confidence in the proposed treatments, potentially hindering their willingness to undergo specific medical procedures.

4) Responsibility Issues in Healthcare Smith *et al.* [78] pointed out that if patients are harmed due to recommendations from an opaque AI system (AIS)

adopted by clinicians, questions arise about how responsibility will be assigned. Specifically, in the healthcare field, doctors are expected to take responsibility for their decisions. If a black-box system provides incorrect recommendations, doctors will find it challenging to explain why they followed the system's advice, potentially raising legal and ethical liability concerns.

Based on the existing studies [79, 80, 81], in the following, we provide the quantification of the loss resulting from the lack of interpretability in black-box models. Specifically, we employ descriptive terms to quantify the degree of loss in two specific scenarios: medical and financial.

- **Medical Scenario** Amann *et al.* [79] pointed out that, in the medical domain, the lack of interpretability in black-box models can lead to serious legal and ethical uncertainty. Without adequate consideration of interpretability, these technologies can neglect regulatory issues and result in significant harm. Moreover, Grote *et al.* [80] pointed out that in the face of a black-box model lacking interpretability, its clinical decision support can constrain the capabilities of physicians. Specifically, physicians can rigidly adhere to the output of the black-box model to avoid being held accountable. This situation poses a serious threat to the autonomy of physicians.
- **Financial Scenario** Yan *et al.* [81] pointed out that, in the financial domain, the lack of interpretability in the decision mechanisms of black-box models poses a challenge for financial practitioners and regulatory authorities in understanding the factors influencing the model's decisions. This can significantly impact the fairness of loan decisions, potentially resulting in substantial financial losses.

Although interpretability is a valuable trait, it is not the sole factor taken into account when deploying models, especially in the healthcare industry [82, 83]. Deep learning has also demonstrated remarkable success in healthcare applications [84]. However, there are compelling reasons that test prioritization for classical models remains highly necessary.

- **Applicability to Structured Medical Data:** Deep learning finds extensive use in the field of medical imaging [83], aiding in the automatic detection of diseases and tumors. However, a substantial portion of data in the healthcare sector exists in structured tabular formats. Classical machine learning models have demonstrated superior performance when dealing with structured medical data, outperforming deep learning methods [85, 86]. For instance, Shwartz *et al.* [85] pointed out that when handling tabular datasets, the classical ML technique XGBoost outperforms the evaluated DL models.
- **Need for Interpretability:** In healthcare [77], when clinicians need to justify their decisions to patients, having an understanding of the reasoning behind model predictions is essential. Classical machine learning models can provide this crucial information [87].
- **Regulatory Approvals:** Regulatory bodies can require models to elucidate the decision-making processes of a model to facilitate comprehensive treatment risk assessment [59]. The interpretability that classical ML models can provide is crucial for obtaining regulatory approvals.

Machine learning testing involves systematically evaluating and validating machine learning models to ensure their accuracy, reliability, and effectiveness in prediction or decision-making [88, 89, 59, 90]. The primary goal is to reveal disparities between intended and actual behaviors exhibited by ML systems [13]. Compared to traditional

software systems, machine learning testing presents distinct challenges. One pivotal challenge is the Oracle Problem [91], which pertains to the difficulty in acquiring accurate labels or ground truth for training and testing data. In the context of testing ML-based systems, automated testing oracles are typically unavailable. Therefore, manual labeling remains the mainstream method, which can lead to substantial labeling costs. In the literature, numerous fields are dedicated to addressing labeling cost concerns, such as test selection [35, 36] and test prioritization [3, 2]. In our study, we concentrate on test prioritization, which will be further elaborated in the subsequent section.

4.2.2 Test Case Prioritization

In the field of traditional software testing, test case prioritization aims to determine the sequence in which test cases are executed to uncover defects more effectively. In the literature, numerous techniques for test prioritization have been proposed. The majority of these approaches are rooted in code coverage analysis. Notably, two primary coverage-based techniques are: Coverage-Total Method (CTM) and Coverage-Additional Method (CAM) [92]. CTM operates by sequentially selecting tests with the highest coverage rates, followed by those with progressively lower rates. In cases where tests share the same coverage rate, the method introduces randomness to determine the prioritization. In contrast, CAM distinguishes itself from CTM by its approach. It strategically utilizes feedback from previous selections, iteratively opting for tests that target previously uncovered code structures, thereby incrementally expanding the coverage.

Test input prioritization in the field of Deep Neural Networks (DNNs) [2, 7, 3, 32] aims to enhance the efficiency of testing by focusing on test inputs that are more likely to expose model misclassifications, thereby revealing potential bugs earlier. This approach ensures that crucial test inputs are identified and labeled promptly within the constraints of limited time. Previous research [3] has indicated that confidence-based approaches outperform the aforementioned coverage-based methods. These confidence-based approaches prioritize tests based on the model's confidence. One notable approach is DeepGini [3], which surpasses all existing coverage-based prioritization methods in terms of both effectiveness and efficiency. A recent comprehensive investigation conducted by Weiss *et al.* delved into the capabilities of various confidence-based DNN test input prioritization techniques, such as Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy. They demonstrated the effectiveness of these approaches in identifying potentially misclassified test inputs.

However, while confidence-based test prioritization methods have been proven effective [3] and can be adapted for classical ML models, their application in the context of test prioritization for classical ML models is hindered by several limitations. We discuss these limitations as follows.

- **Single dimension on binary classification models** Binary classification models [93, 94] categorize test inputs into two distinct classes, which limits the application of confidence-based test prioritization approaches to a single dimension. Specifically, when applying confidence-based approaches to these models, the first step is calculating the probabilities for each classification, denoted as $(p, 1-p)$. If the model's prediction probability for a test is $(0.5, 0.5)$, it means the model is most uncertain about this test [3], indicating this test is more likely to be misclassified. The closer a test's p value is to 0.5, the more uncertain the model

is about that particular test. Consequently, uncertainty is solely determined by p. Regardless of the specific confidence-based test prioritization method employed, tests with p values closer to 0.5 will be prioritized over others. To illustrate this point, consider a hypothetical test set with three tests, and the model's probability vectors for these tests are as follows: t_1 $(0.9, 0.1)$, t_2 $(0.7, 0.3)$, t_3 $(0.8, 0.2)$. Irrespective of the chosen confidence-based test prioritization method, the resulting ranking will be $t_2 \rightarrow t_3 \rightarrow t_1$ because t_2 has the p value (0.7) closest to 0.5, followed by t_3 (p = 0.8), while t_1 has the farthest p value from 0.5 (p = 0.9).

The above conclusions have been confirmed through our experimental results. For each subject, all confidence-based methods yield identical effectiveness, indicating they produce the same ranking for a given test set.

- **Lack of model-specific insights** Confidence-based approaches for test prioritization consider the model a black box and rely solely on its prediction probability vectors. This neglects the transparency and interpretability of classical ML models, which are mostly white-box and have an understandable decision-making process. As a result, confidence-based approaches fail to incorporate crucial model-specific insights from classical ML models, leading to suboptimal test prioritization.
- **Ignoring attribute features** Furthermore, confidence-based approaches ignore a crucial aspect of the test datasets for classical ML models, namely, the attribute features. These features are carefully engineered by domain experts to effectively capture and represent crucial aspects of the underlying data. They can directly reflect the attribute information of each test input. However, confidence-based approaches ignore this crucial feature information in the process of test prioritization.

To overcome the aforementioned limitations, we propose MLPrior, a test prioritization approach specifically tailored for classical ML models. MLPrior leverages the characteristics of classical ML classifiers (i.e., interpretable models and carefully engineered attribute features) to prioritize test inputs. The core premises behind MLPrior are twofold: 1) tests more sensitive to the injected mutations are more likely to reveal bugs, and 2) test inputs closer to the decision boundary of the model are more likely to be predicted incorrectly.

The first premise is grounded in the well-established practice of traditional mutation testing [62, 63, 64, 95, 96], which considers that test cases sensitive to mutations (able to capture mutants) have a higher capability to detect bugs in software. The second premise has been identified and demonstrated in prior work [48].

4.2.3 Mutation Testing

Mutation testing [38, 39] is a systematic software testing technique that has gained significant attention in both academic and industrial research communities [97, 98]. The fundamental principle is to introduce small and intentional modifications, called mutants, into the source code of a software system [99]. These mutations simulate potential faults that may occur during the execution of the program. A well-designed test suite should be able to detect the presence of these mutants, indicating its capability to detect real faults in the code [40]. In the context of mutation testing, the term "kill" refers to the ability of a test case to detect a specific mutant [100]. When a test case "kills" a mutant, it means that the test case is able to reveal a difference in behavior between the original program and the mutated version of the

program. A test suite with a high "mutation kill" rate is considered more effective and reliable, as it demonstrates a greater ability to detect potential faults or deviations from the expected behavior.

4.2.4 Automated Labeling Approaches for Machine Learning

Data labeling is a labor-intensive task that is indispensable in the development of supervised machine learning systems [101]. Conventional data labeling methods typically rely on manual effort, which is a time-consuming and costly process. Moreover, in specialized fields like medicine and finance, manual labeling necessitates domain-specific expertise, further increasing its cost. In recent years, various automated or semi-automated data labeling methods [102, 103] have emerged, aimed at reducing the burden of manual labeling and improving the overall labeling efficiency.

Desmond *et al.* [103] introduced a semi-automated data labeling system that views the labeling task as a collaborative effort between human annotators and machine annotators, which are implemented as predictive models. The core of this approach involves a human-machine coactive process facilitated by a semi-supervised predictive model and an active learning selector. In each iteration, the active learning selector prioritizes the most uncertain examples for annotation by human annotators based on the model's predictions. The consistency between human decisions and machine predictions is continuously monitored and presented at various checkpoints, allowing annotators to assess the machine's performance in the labeling task. Once annotators are satisfied with the machine's performance, they can delegate the remaining labeling tasks to the machine (automatic labeling).

Wu *et al.* [104] proposed a semi-automated labeling method based on active learning and label informativeness. Specifically, the SLMAL algorithm selects the most informative example-label pairs for annotation by combining the uncertainty of examples and the informativeness of labels. During this process, the algorithm initially identifies and prioritizes the example-label pairs in need of labeling the most and subsequently employs the nearest neighbors of these highly uncertain pairs to predict their partial labels.

However, semi-automatic labeling comes with several limitations:

- **Human Involvement:** In the semi-automatic labeling process, human intervention is still required, especially in complex decision-making processes. This can result in an increase in overall labeling time and costs, particularly in situations requiring domain expertise.
- **Scalability:** Semi-automatic labeling methods can face challenges when dealing with large-scale datasets, primarily regarding processing speed and resource utilization.
- **Sensitivity to Labeling Quality:** The performance of the model is largely dependent on the quality of the initial labeled data used for training. Low-quality or biased labeling data may lead to a decrease in model performance.

However, despite the presence of semi-supervised learning, in order to labeling tests more accurate and of higher quality, manual labeling is still the mainstream in the industry [102].

Automated labeling methods offer a potential solution to the aforementioned limitations. Nevertheless, due to the constraints outlined below, fewer automated labeling methods are specifically designed for classical machine learning. To our best knowledge, the known method applicable to labeling for classical machine

learning models is Programmatic Labeling [105]. Programmatic labeling automates the labeling process through scripts and programming algorithms, significantly improving the efficiency of data preparation. However, Programmatic Labeling typically requires specialized programming skills to create labeling rules, which may pose a barrier for researchers without a technical background.

In the following, we outline the challenges that make it difficult to develop automated labeling methods specifically designed for classical machine learning, resulting in the current scarcity of such methods.

- **Diversity of Domain Knowledge** Automated labeling methods face challenges in accommodating diverse types of datasets, each requiring expertise from different domains. For example, a social network dataset can involve knowledge from linguistics, psychology, and sociology, while a medical dataset, such as cancer data, requires expertise in medicine and biology. The intricate and extensive nature of knowledge across different fields presents a challenge in developing a universally applicable automated labeling technique.
- **Domain Adaptation Challenges** Even within the same domain, different tasks may necessitate varying areas of expertise. For instance, in cancer research, labeling data for different types of cancers (such as lung cancer, breast cancer, etc.) can require specialized medical knowledge and skills.
- **Difficulty in Quantifying Domain Knowledge** Encoding domain expertise into an automated labeling system can be a complex task.

4.3 Approach

4.3.1 Overview

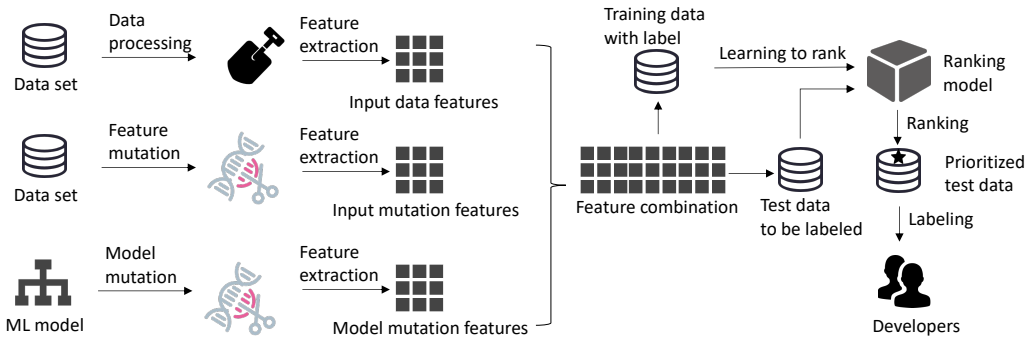


Figure 4.1: Overview of MLPrior

In this paper, we propose MLPrior, a test prioritization approach specifically designed for classical ML models. Figure 4.1 illustrates the workflow of MLPrior. Given a test set T and an ML model M, MLPrior produces a sorted test set T', where test cases that are more likely to be mispredicted by the model are placed at the front. We outline the steps of MLPrior as follows.

❶ **Attribute feature generation:** In the initial stage, MLPrior converts the attribute values of each test $t \in T$ into a feature vector, denoted as V_t^D. This involves transforming non-numeric attributes into a numeric format. To accomplish this, we create a mapping dictionary that includes all non-numeric attributes paired with their corresponding numeric values. For instance, in the context of the attribute "gender," the values "male" and "female" are mapped to 0 and 1, respectively.

❷ **Mutation feature generation (model):** Based on the model mutation rules

described in Section 4.3.2, MLPrior generates a set of mutated models for the original ML model M. For each test $t \in T$, MLPrior identifies whether t "kills" each of the mutated models (i.e., whether the predictions made by the mutated model and the original ML model for t are different). This process allows MLPrior to construct a model mutation feature vector, denoted as V_t^M. Each element of V_t^M corresponds to a specific mutated model. More specifically, MLPrior sets the i_{th} element of t's model mutation vector to 1 if t kills the i_{th} mutated model. Otherwise, the element is set to 0.

❸ **Mutation feature generation (inputs):** Based on the input mutation rules outlined in Section 4.3.2, MLPrior generates mutated inputs for each test instance $t \in T$. By comparing the predictions of model M on the i_{th} mutated input with its predictions on the original test input t, MLPrior constructs an input mutation vector denoted as V_t^I. If the prediction of model M for the i_{th} mutated input is different from that of the original test input t, the i_{th} element of V_t^I is set to 1. Otherwise, it is set to 0.

❹ **Feature Concatenation:** For each test $t \in T$, MLPrior concatenates the three types of feature vectors constructed in the previous steps (i.e., V_t^D, V_t^M and V_t^I) and obtain a final feature vector, denoted as V_t.

❺ **Learning-to-Rank:** For each test instance $t \in T$, MLPrior feeds its final feature vector (V_t) into the pre-trained XGBoost ranking model [8], which will produce the probability of this input being misclassified. Finally, MLPrior ranks all the tests in T based on their probability scores in descending order, thereby prioritizing the possibly-misclassified tests.

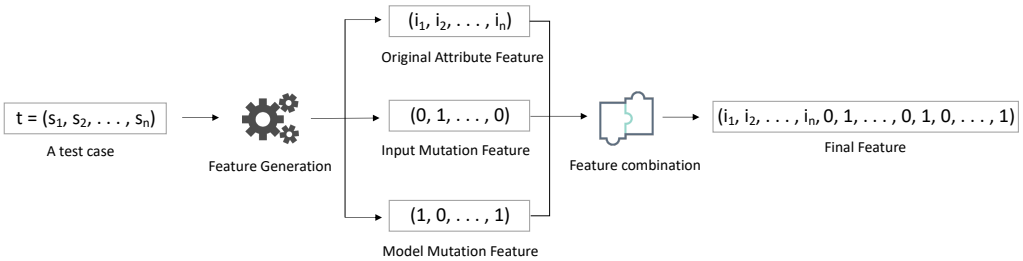


Figure 4.2: A concrete example of feature generation in MLPrior

In MLPrior, the concept of **feature** is crucial. To demonstrate the processes of feature extraction, combination, and concatenation more intuitively, we provide a typical example. In this example, we delve into the specifics of how MLPrior generates features for a given test t, illustrating each step of the process in detail. Furthermore, we visually illustrated this example in Figure 4.2 to enhance the presentation of MLPrior's feature generation process.

- **Feeding Attributes of t to MLPrior** Given a classical ML model M and its corresponding test set T, let t be a test instance from the test set T. Given that the dataset for the classical ML model is in a tabular format, we assume the attribute features of t as $t = (s_1, s_2, \ldots, s_n)$. Here, s_n includes both numeric and non-numeric formats (such as strings). In this step, we input the attributes of the test t into MLPrior.
- **Generation of Original Attribute Features** We input the attribute vector of test t, which is $(s_1, s_2, \ldots, s_n)$, into MLPrior. MLPrior then converts all non-numeric attributes into numeric format to construct the original attribute vector of t, represented as $(i_1, i_2, \ldots, i_n)$.
- **Generation of Input Mutation Features** Subsequently, MLPrior generates N

mutated inputs of the test t, denoted as $(t_1, t_2, \ldots, t_N)$. MLPrior then feeds these mutated inputs into the original ML model to make predictions. If the model output for t_i differs from the result of the original sample t, the i_{th} element of the input mutation feature vector will be set to 1; otherwise, it will be set to 0. In this manner, we obtain the input mutation feature vector for t, represented as $(0, 1, \ldots, 0)$. This vector indicates that for the first mutant of t, denoted as t_1, the model's prediction is the same as for t. For the second mutant of t, denoted as t_2, the model's prediction differs from that for t.

- **Generation of Model Mutation Features** For the original model M, MLPrior generates K mutated models, denoted as $(m_1, m_2, \ldots, m_K)$, and inputs the original sample t into these mutated models for prediction. If the prediction of the i_{th} mutated model for t differs from the prediction of the original model M for t, then the i_{th} element of the model mutation feature vector is set to 1; otherwise, it is set to 0. Through this method, we obtain the model mutation feature vector for t, represented as $(1, 0, \ldots, 1)$. This vector indicates that the first mutated model of M, denoted as m_1, predicts differently for the test t compared to the original model M. Conversely, the second mutated model of M, denoted as m_2, predicts the same for the test t as the original model M.
- **Feature Combination**: MLPrior concatenates the three types of features obtained from the previous steps (i.e., Original Attribute Features, Input Mutation Features, and Model Mutation Features) to form the final feature vector for t. This final feature vector is represented as (Original Attribute Features, Input Mutation Features, Model Mutation Features) $= (i_1, i_2, \ldots, i_n, 0, 1, \ldots, 0, 1, 0, \ldots, 1)$.

The primary purpose of this step is to encapsulate the attribute information of each test instance $t \in T$ into a feature vector, which will then be utilized as input to the ranking models for test prioritization. Since the ranking models require numeric inputs, MLPrior converts all the non-numeric attribute values of t into a numeric format. To this end, we construct a mapping dictionary that specifies the numeric value corresponding to each non-numeric attribute value. For instance, for the attribute "gender," the attribute value "male" is transformed into 1, while "female" is transformed into 0. The motivation behind extracting these original features is explained as follows.

Prior research [48, 106] pointed out that *test inputs situated closer to the decision boundary of a model are more likely to be misclassified*. In order to effectively capture the spatial relationship between a test input and the decision boundary and to preserve the carefully-selected and low-dimensional features of the classical ML test set, we directly generate feature vectors of each input from its original attribute values.

4.3.2 Mutation Rule Specification

In this stage, we propose two types of mutation rules designed specifically for classical ML models and their corresponding datasets. The principle underlying our utilization of mutation testing in test prioritization is: *If a test input exhibits high sensitivity to the injected mutations, this input is more likely to detect faults in the system*. This principle is derived from previous research in traditional mutation testing [64, 95, 96]. We extend this principle to encompass ML systems, correspondingly designing model mutation rules and input mutation rules. The key insights of MLPrior are that: 1) if an input can kill many mutated models (i.e., the predictions

for the input made by the mutated models and the original model are different), indicating that this input is sensitive to model mutations, MLPrior considers this input more likely to be misclassified. 2) If the prediction result for a given test input is different from that of many of its mutated inputs, indicating that the predictions for the input are sensitive to the mutations, MLPrior considers this input more likely to be misclassified. In the following sections, we provide a detailed explanation of our mutation approaches.

4.3.2.1 Model mutation rules

The model mutation rules are designed to make slight changes to the architecture parameters or weight parameters of the pre-trained ML models to generate mutated models. We ensure that the new parameter values are close to their original values in order to achieve slight mutations. It is important to note that this process does not involve any retraining operations. Therefore, the total execution time of generating model mutants is short, with an average duration of 3 seconds, as shown in Table 4.4.

In our study, we evaluated the effectiveness of MLPrior using five classical ML models, namely Decision Tree [9], K-Nearest Neighbors (KNN) [107], Logistic Regression (LR) [14], XGBoost [8], and Gaussian Naive Bayes (GaussianNB) [8]. The rationale behind selecting these models is twofold: 1) They have gained widespread adoption in various industries due to their interpretability and proven performance [72, 108]; 2) These models have been extensively utilized in recent ML testing studies [66]. It is important to note that MLPrior's applicability extends beyond the evaluated models. By making simple adjustments to the model mutation rules (i.e., enabling them to target the architecture parameters or weight parameters of the evaluated model), it can be adapted to a diverse range of interpretable ML models. We elaborate on the specific details of conducting model mutation as follows.

❶ **Decision Tree** [9] Decision tree is a machine learning method that predicts data step-by-step based on features. During prediction, attribute values are utilized to make decisions at internal nodes of the tree, determining which branch node to enter based on the decision outcome until a leaf node is reached to obtain the classification result.

Input to Decision Tree: The input to a Decision Tree consists of a dataset containing instances with associated features. The Decision Tree algorithm utilizes these input features to create a hierarchical structure that facilitates effective classification.

Process of Classification: Decision tree operates by sequentially making decisions at each split node of the tree. For a given input, it begins at the root node and evaluates the features of the input to determine the appropriate branch to follow at each split node. This process iterates until a leaf node is reached, signifying a classification outcome.

Mutating Decision Tree: To induce mutation in the Decision Tree model, we randomly select a set of split nodes and introduce random deviations to their threshold values, thereby influencing the predictive outcomes of the Decision Tree model. We explain below why changing the thresholds can alter the predictive results of a decision tree: Consider a situation where a given test sample t passes through nodes in the original tree. Based on decisions made at split nodes, it arrives at leaf node A, and thus will be classified as A category. After making

slight adjustments to the thresholds of a group of decision nodes, when sample t traverses the mutated tree, the modified decision thresholds at split nodes can lead it to reach leaf node B.

❷ **K-Nearest Neighbors (KNN)** [107] KNN (K-Nearest Neighbors) is a widely adopted classical machine learning model. It classifies an input into a class based on the majority class of its K nearest neighbors in the feature space.

The parameter K: The parameter K represents the number of neighbors considered. For example, when the value of K is 8, it means that when predicting the label or value of a new data point, the algorithm will find the eight closest samples from the training data and then determine the classification of the sample based on the classification of these neighboring samples. The choice of the value of K affects the complexity and performance of the model.

Mutating KNN: To induce mutations in the KNN model, we introduce a random slight alteration to the value of K, thereby influencing the prediction outcomes. For instance, consider an initial K value of 8 for a KNN algorithm. Given a sample t, the KNN model's prediction for t is determined based on the categories of its nearest 8 neighbors. Assuming among these 8 neighbors, 5 belong to category A and 3 belong to category B, the final classification for t would be A. If K is slightly perturbed, changing it to 12, and the newly added neighbors all belong to category B, then in this scenario, among the 12 nearest neighbors, 7 belong to category B and 5 belong to category A, resulting in the final classification for t being B. Thus, variations in the value of K can introduce disturbances in model prediction results.

❸ **Logistic Regression (LR)** [14] Logistic regression establishes a linear functional relationship to construct a connection between input features and probability outputs. It employs a Sigmoid function (as displayed in Formula 4.1) to map the results onto the [0, 1] interval, representing the probability of belonging to class 1. This enables the classification of input samples.

Weight Coefficient: In the Sigmoid function of logistic regression, weight coefficients determine the impact of different features on predicting the output. Each feature is assigned a corresponding weight coefficient. For example, in Formula 4.1, the weight coefficient for the feature x_0 is w_0.

Mutating Logistic Regression: To introduce mutation to the Logistic Regression, we randomly select a feature from the Sigmoid function and modify its weight coefficient, thus affecting the model's predictions. For example, consider Formula 4.1, which represents a trained Logistic Regression model taking four input features: x_0, x_1, x_2, and x_3. In this formula, w_0, w_1, w_2, and w_3 denote the weight coefficients for each feature, and $f(x)$ represents the prediction score. We mutate the model by randomly selecting one of the four weight coefficients and making a slight adjustment to its weight coefficient. This mutation process directly influences the output value of $f(x)$, consequently impacting the classification results of the model.

$$f(x) = \frac{1}{1 + e^{-(w_0x_0+w_1x_1+w_2x_2+w_3x_3+b)}} \tag{4.1}$$

❹ **XGBoost** [8] XGBoost is a widely used gradient boosting algorithm designed for enhanced predictive modeling. XGBoost is a variant of the boosting algorithm [109], which aims to integrate multiple weak classifiers into a robust classifier.

As a boosting tree model, XGBoost aggregates multiple tree models to form a powerful classifier. In binary classification tasks, XGBoost defaults to output 0 or 1, representing two different classes. Internally, XGBoost calculates an initial probability value p, subsequently comparing it to a threshold (with a default value of 0.5) prior to determining the final class output: values exceeding 0.5 yields an output of 1, whereas values below 0.5 yield an output of 0.

Mutating XGBoost: To mutate XGBoost, we apply a random slight offset to the internal threshold of the XGBoost model, thereby generating model mutants. For instance, consider the original XGBoost threshold of 0.5; upon introducing a minor offset, the threshold becomes 0.4 for the mutated XGBoost model. Under this mutation, the following scenarios arise: 1) Given a test input t_1 with a predicted p value of 0.45, the original XGBoost predicts an outcome of 0 (p <0.5), whereas the mutated XGBoost predicts an outcome of 1 (p >0.4); 2) Given another test input t_2 with a p value of 0.3, both the original XGBoost and the mutated XGBoost models predict an outcome of 1 (p >0.4; p >0.5).

It can be observed that t_1 is more sensitive to the injected mutation than t_2, and we consider that t_1 is more likely to be misclassified by the model. This mutation rule can be reasonably interpreted from an uncertainty perspective: when a slight adjustment in the model's classification threshold can alter the test's classification result, it indicates that the model's prediction probability for that test is close to 0.5. According to prior work [3], the closer a prediction probability is to 0.5, the greater the model's uncertainty regarding that test, making it more prone to misclassification.

❺ **Gaussian Naive Bayes (GaussianNB)** [8] Gaussian Naive Bayes (GNB) is a probabilistic machine learning classification technique based on Gaussian distribution. It assumes that each parameter (a feature) possesses independent predictive power for the output variable. The combination of predictions from all parameters yields the final prediction.

Mutating GaussianNB: To induce mutations in GaussianNB, we introduce a random slight adjustment to the internal threshold of the GaussianNB model, resulting in the generation of model mutants.

4.3.2.2 Input mutation rules

The prior work [48] introduced a mutation operator, *noise perturbation*, for mutating inputs in image format, which adds noise to data for mutation. A common type of image noise is occlusion noise [110], achieved by overlaying a black block on the part of the image. This black block typically consists of a matrix filled with 0. The method involves replacing the matrix of pixel values at the original location in the image with this zero-filled matrix (black block). Inspired by this technique, MLPrior's input mutation rule involves randomly selecting a specific feature from the feature vector of t and changing its value to 0. Before this, MLPrior initially converts all attributes of t into a corresponding numerical feature vector. The objective is to alter the attribute value of this particular feature, thus affecting the model's predictions. To gain a deeper insight into the impact of input mutation rules on model predictions, we provide explanations using the five classical ML models evaluated in our study as examples. It is important to note that our input mutation rules are applicable to a wide range of datasets for classical ML models.

❶ **Decision Tree** Given a test input, if a specific feature value of this input is changed to 0, it could lead to a change in the decision path that the input takes down the tree. This mutation can cause the input to be categorized differently than it would have been without the mutation.
❷ **K-Nearest Neighbors (KNN)** For KNN, changing the value of a feature to 0 can alter the distance calculation between this input and other instances. This shift in distances can lead to a different set of k nearest neighbors being considered, thereby potentially affecting the classification result of the input.
❸ **Logistic Regression** In logistic regression, modifying a feature's value to 0 will impact the coefficients associated with that feature. This can lead to a different logistic function, causing the instance's predicted probability to shift, ultimately affecting the classification outcome.
❹ **XGBoost** For a given sample, setting a feature of it to 0 can influence the way that features contribute to the ensemble of decision trees. This can lead to different tree structures being emphasized during prediction, thereby affecting the final prediction of the sample.
❺ **Gaussian Naive Bayes (GaussianNB)** For a given sample, setting a feature's value to 0 can impact the calculation of probabilities for the various classes based on the Gaussian distribution assumption. This can influence the final classification result.

4.3.3 Mutation Feature generation

For each test $t \in T$, based on the aforementioned mutation rules, we generate mutants and subsequently build mutation feature vectors. The detailed procedures are elaborated below.

- **Input Mutation Features (IMF)** Based on the input mutation rules presented in Section 4.3.2.2, MLPrior generates a set of input mutants for each test $t \in T$. Subsequently, MLPrior proceeds to compare the predictions of model M for each input mutant with that of the original input t to construct the input mutation vector. During this process, if the prediction for the i-th mutated input differs from that of the original test input t, the corresponding i-th element of the feature vector is assigned a value of 1; otherwise, it is assigned a value of 0. An example of the resulting feature vector is (0, 1, ..., 0).
- **Model Mutation Features (MMF)** Based on the model mutation rules described in Section 4.3.2, MLPrior generates a set of mutated models for the original ML model M. For each test $t \in T$, MLPrior identifies whether t "kills" each of the mutated models (i.e., whether the predictions made by the mutated model and the original ML model for t are different) to construct the model mutation vector. More specifically, if t kills the i-th mutated model, the i-th element of t's model mutation vector will be set to 1. Otherwise, the i-th element will be set to 0. An example of the resulting feature vector is (1, 0, ..., 0).

4.3.4 Feature Concatenation

Based on the aforementioned steps, for each test sample $t \in T$, MLPrior generates three types of feature vectors: the attribute feature vector, the input mutation vector, and the model mutation vector. Subsequently, for $t \in T$, MLPrior concatenates these three types of features to obtain the final feature vector, which is then used as input to the ranking model.

4.3.5 Learning-to-rank

Once obtaining the feature vector for each $t \in T$, MLPrior aims to train a ranking model to automatically learn the probability of a test input t being misclassified by the ML model M based on its feature vector. In the following section, we describe the process of constructing the ranking model and explain how to utilize the ranking model for test prioritization.

Ranking model building MLPrior leverages the XGBoost ranking algorithm [8], an optimized distributed gradient boosting learning algorithm, to construct the ranking model. Given the classical ML model M with dataset D, we first split the dataset D into two partitions: the training set R and the test set T, in a 7:3 ratio [111]. The test set remains untouched for the purpose of evaluating MLPrior. Based on the training set R, our objective is to construct a training set R' for training the ranking models. To achieve this, we generate the final feature vector for each $r \in R$, following the steps described in Section 4.3.2 to Section 4.3.4. These features are used as the training features for the dataset R'. Next, we utilize the original ML model M to classify each instance $r \in R$ and then compare the model's predictions with the corresponding ground truth of r. By doing so, we can identify whether r is misclassified by the model M. If r is misclassified, we label it as 1; otherwise, we label it as 0. As a result, we obtain the labels for the training set R'. Based on the constructed training set and corresponding training labels obtained above, we can proceed to train the ranking model of MLPrior.

Test prioritization via ranking model It is essential to emphasize that the XGBoost ranking algorithm, upon completion of its training process, is a binary classification algorithm. It classifies a test into two categories instead of providing an estimation of misclassification probability. Therefore, we made specific adjustments to the original XGBoost algorithm. Specifically, we extract the intermediate value from the model's output, which was originally used to determine whether a test instance would be predicted incorrectly or not. Typically, if the intermediate value surpasses the threshold, the input is classified as "misclassified"; otherwise, it is classified as "not misclassified". Instead of proceeding with the final classification, we directly employ this intermediate value as the misclassification probability score. A high value denotes that a test instance has a high probability of being misclassified. Finally, we sort all the tests in the test set T in descending order based on their misclassification probability scores, resulting in the prioritized test set T'.

4.3.6 Variants of MLPrior

In order to explore the influence of different ranking models on the effectiveness of MLPrior, we propose four variants, denoted as MLPrior^T, MLPrior^K, MLPrior^L, and MLPrior^N. These variants utilize different ranking models for test prioritization, namely, decision tree [112], K-nearest neighbors (KNN)[69], logistic regression[14], and Gaussian Naive Bayes (GaussianNB) [113], respectively. They solely differ in the selection of the ranking models, while the remaining workflow is kept identical.

- **MLPrior^T** This variant incorporates the decision tree ranking model. The principle of the Decision Tree algorithm is to partition the dataset into subsets at split nodes, iteratively branching until reaching leaf nodes that provide the final classification.
- **MLPrior^K** integrates the KNN algorithm. KNN is a well-established machine learning technique. It operates on the fundamental principle of proximity, where

the classification of a sample is determined by considering the majority labels of its K nearest neighbors in the feature space.
- **MLPrior**^{L} integrates the Logistic Regression algorithm [14]. Logistic Regression employs the logistic function to transform the linear combination of the independent variables into a range between 0 and 1. Consequently, this probability value is employed to perform classification.
- **MLPrior**^{N} integrates the Gaussian naive Bayes (GaussianNB) ranking model. GaussianNB is a probabilistic machine learning classification technique based on the Gaussian distribution. It assumes that each feature possesses independent predictive power for the output variable. The final prediction is obtained by combining the predictions derived from all features.

4.4 Study Design

4.4.1 Research Questions

Our experimental evaluation answers the research questions below.

- **RQ1: How does MLPrior perform in terms of effectiveness and efficiency?**
  To solve the labelling cost problem, we propose MLPrior, a test input prioritization approach specifically designed for classical ML models. In this research question, we evaluate the effectiveness and efficiency of MLPrior by comparing it with several existing test prioritization approaches [3, 7].
- **RQ2: How does MLPrior perform on different types of test inputs?**
  In order to evaluate the effectiveness of MLPrior in various scenarios, we constructed mixed noisy datasets and fairness datasets. We compare the effectiveness of MLPrior against various test prioritization approaches on the generated datasets.
- **RQ3: How do different ranking algorithms impact the effectiveness of MLPrior?**
  In MLPrior, we employ XGBoost [8] as the ranking model for test prioritization. In this research question, we investigate the impact of different ranking models on the effectiveness of MLPrior. To this end, we construct four variants employing different ranking models: decision tree [112], K-nearest neighbors (KNN)[69], logistic regression[14], and Gaussian Naive Bayes (GaussianNB) [113]. By evaluating the effectiveness of these variants, we explore the influence of ranking models.
- **RQ4: To what extent does each type of features contribute to the effectiveness of MLPrior?**
  To construct the feature vector for a given test input, MLPrior generates three types of features: model mutation features, dataset mutation features, and attribute features. In this research question, our objective is to investigate the extent to which each type of features contributes to the effectiveness of MLPrior.
- **RQ5: How does the selection of main parameters of MLPrior impact its effectiveness?**
  We investigate the influence of the main parameters in MLPrior. Our objective is to evaluate whether MLPrior can consistently outperform the compared test prioritization approaches when these main parameters fluctuate.

4.4.2 Subjects

In our research, we utilized 305 subjects to assess the effectiveness of MLPrior. A subject in this context refers to a combination of a classical ML model and a dataset. The description of these subjects can be found in Table 4.1. Out of the 305 subjects, 25 subjects (5 datasets × 5 ML models) were generated using natural datasets, while 250 subjects were generated using mixed noisy datasets. Additionally, 30 subjects were generated using fairness datasets. Below, we explain the construction method for mixed noisy datasets and fairness datasets.

- **Mixed noisy datasets** blend natural data with noisy data, with the natural data accounting fo r 70% and the noisy data accounting for 30%. The reason we chose 30% is that: A high noise ratio, such as 90%, would lead to a substantial proportion of noisy test inputs. In this scenario, a significant number of misclassified tests would be chosen by any prioritization method, making it difficult to demonstrate the effectiveness of MLPrior. Therefore, to ensure an effective evaluation of MLPrior and the compared approaches, we choose a reasonable noise generation ratio (i.e., 30%). For each of the five natural datasets, we generated 10 mixed noisy datasets, resulting in a total of 50 (5 × 10) mixed datasets. Each mixed dataset was paired with five classical ML models, leading to 250 subjects (50 datasets × 5 models).
- **Fairness datasets** refer to datasets carefully constructed with a specific focus on avoiding the introduction of biases related to individual attributes, such as gender, age, etc. In our study, we generated a fairness dataset from a natural dataset following the approach utilized in prior work [66]: we randomly selected a subset of instances and modified their gender and age attribute values while keeping their original labels untouched. Employing this approach, we generated 6 fairness datasets. We pair each dataset with five classical ML models, leading to 30 subjects (6 datasets × 5 models).

4.4.2.1 Datasets

In our study, we evaluate MLPrior using five datasets: Adult [114], Bank [115], Stroke [116], Diabetes [117] and Heartbeat [118]. The reason for selecting these five datasets lies in their widespread utilization in the field of machine learning. Moreover, these datasets have been extensively employed in multiple recent research on classical machine learning testing, including FSE 2022 [66] and ICSE 2022 [119, 120, 121].

- **Adult** [114, 122, 123]: The adult dataset is designed to predict whether an individual's annual income exceeds 50K based on various demographic and financial attributes. It consists of 48,842 instances, with each instance representing a single individual. All the instances are divided into two classes: >50K and <=50K. Each individual is described by 14 different attributes, such as age, occupation, education level, workclass, etc.
- **Bank** [115, 122]: The bank dataset is utilized to forecast whether a client will subscribe to a term deposit, utilizing their demographic, financial, and social information. It consists of 49,732 instances, classified into two classes: subscribing to the term deposit or not subscribing. Each instance encompasses 16 attributes, such as age, education, loan, and balance.
- **Stroke** [116]: The stroke dataset is employed for predicting the occurrence of a stroke in patients. It comprises 40,907 instances, classified into two classes: having

Table 4.1: Classical ML models and datasets

ID	Datasets	# Size	Models	Type
1	Adult	48,842	Tree	Original, Noisy, Fairness
2	Adult	48,842	KNN	Original, Noisy, Fairness
3	Adult	48,842	LR	Original, Noisy, Fairness
4	Adult	48,842	NB	Original, Noisy, Fairness
5	Adult	48,842	XGB	Original, Noisy, Fairness
6	Bank	49,732	Tree	Original, Noisy, Fairness
7	Bank	49,732	KNN	Original, Noisy, Fairness
8	Bank	49,732	LR	Original, Noisy, Fairness
9	Bank	49,732	NB	Original, Noisy, Fairness
10	Bank	49,732	XGB	Original, Noisy, Fairness
11	Stroke	40,907	Tree	Original, Noisy, Fairness
12	Stroke	40,907	KNN	Original, Noisy, Fairness
13	Stroke	40,907	LR	Original, Noisy, Fairness
14	Stroke	40,907	NB	Original, Noisy, Fairness
15	Stroke	40,907	XGB	Original, Noisy, Fairness
16	Diabetes	253,680	Tree	Original, Noisy, Fairness
17	Diabetes	253,680	KNN	Original, Noisy, Fairness
18	Diabetes	253,680	LR	Original, Noisy, Fairness
19	Diabetes	253,680	NB	Original, Noisy, Fairness
20	Diabetes	253,680	XGB	Original, Noisy, Fairness
21	Heartbeat	30,000	Tree	Original, Noisy, Fairness
22	Heartbeat	30,000	KNN	Original, Noisy, Fairness
23	Heartbeat	30,000	LR	Original, Noisy, Fairness
24	Heartbeat	30,000	NB	Original, Noisy, Fairness
25	Heartbeat	30,000	XGB	Original, Noisy, Fairness

a stroke or not having a stroke. Each instance is described using 10 attributes, such as age, heart disease, hypertension, work type, residence type, and smoking status.

- **Diabetes** [117]: The diabetes dataset is utilized for predicting diabetes occurrence in patients. It comprises 253,680 survey responses related to diabetes. This dataset is categorized into three classes: 0 for no diabetes or diabetes only during pregnancy, 1 for prediabetes, and 2 for diabetes.
- **Heartbeat** [118]: The Heartbeat dataset is used for classifying heartbeat signals. In our experiments, we used 30,000 heartbeat signal sequence data. Each sample in the dataset has a consistent sampling frequency and equal length in its signal sequence. The Heartbeat dataset is divided into 4 classes, which are categorized as heartbeat signal types (0, 1, 2, 3).

4.4.2.2 Classical ML models

We evaluate the effectiveness of MLPrior using five well-established classical ML models: Decision Tree [9], K-Nearest Neighbors (KNN) [107], Logistic Regression (LR) [14], XGBoost [8], and Gaussian Naive Bayes (GaussianNB) [8]. These models were chosen based on two primary reasons: First, their widespread adoption in various industries owing to their interpretability and demonstrated performance [1, 72, 108, 124].

In the industry, the five classical ML models we evaluated are broadly implemented, and their accuracy is crucial, as their prediction errors could have serious consequences. Therefore, thorough testing and test prioritization of these classical ML models are essential.

- **Hospitality industry** [108] The **logistic regression** model can utilize financial data to predict whether a hotel business is at risk of bankruptcy. Investors in the hotel industry will rely on these models to make crucial financial and operational decisions. If the predictions are inaccurate, Investors can make erroneous investment decisions, such as investing in businesses that are at risk of bankruptcy.
- **Service industry** [72] The **decision tree** model can be employed to analyze the impact of information and communication technology (ICT) on service industry performance using global service industry data from the World Bank. Service industry companies will depend on such analyses to formulate strategies, such as investing in ICTs. Incorrect predictions could result in misallocation of resources, affecting the company's long-term performance and competitiveness.
- **Financial industry** [125, 126] The **XGBoost** algorithm can be utilized for personal credit risk assessment. Rao *et al.* [125] employed XGBoost to predict an individual's credit risk for determining loan approval decisions. Moreover, **KNN** can be used for credit scoring(i.e., assessing the credit risk of loan applications) [126].
- **Healthcare industry** [127] The **Gaussian Naive Bayes** model can be leveraged for diagnosing cancer based on the patient's medical information [127].

To better illustrate the utility of MLPrior, we provided a specific example. For instance, in the above scenario where XGBoost is used for personal credit risk assessment, MLPrior can be utilized to identify misjudged loan approvals (where the XGBoost model incorrectly classifies some applicants who should not receive loans as qualified borrowers, thus approving their loan applications). This enables financial institutions to detect and focus on potential high-risk cases earlier, thereby not only reducing losses but also enhancing their overall efficiency in risk management.

Second, their extensive use in recent ML testing studies [66, 128, 129, 130, 131]. Importantly, it should be noted that MLPrior's applicability is not limited to the evaluated models. With minor adjustments to the model mutation rules (i.e., making them target the architecture parameters or weight parameters of the assessed ML model), MLPrior can be adapted to various interpretable ML models.

- **XGBoost** [8] XGBoost, an ensemble method that belongs to the family of boosting algorithms, functions by integrating the forecasts of multiple Classification and Regression Trees (CART) [132] to create a robust classification mechanism. This algorithm amalgamates weak learners to engineer a powerful model with superior predictive capacity.
- **Gaussian Naive Bayes (GaussianNB)** [127] Gaussian Naive Bayes, a probabilistic classifier based on Bayes' theorem with an assumption of independence among predictors, is known for its efficacy in multiclass classification problems and its robustness against irrelevant features.
- **Logistic Regression (LR)** [14] Logistic Regression is a widely-adopted statistical model employed in scenarios of binary classification tasks. This model is founded on the principles of probability and logistic function, offering an interpretable mathematical framework.
- **Decision Tree** [9] Decision tree constructs a tree-like structure, where internal nodes represent decision points based on feature values, and leaves represent the predicted outcomes.
- **K-nearest neighbors (KNN)** [107] KNN is a widely-adopted classification

algorithm that assigns labels to instances based on the majority vote of their K neighboring data points. The KNN algorithm is known for its simplicity and flexibility in handling classification tasks.

4.4.3 Compared Approaches

To demonstrate the effectiveness of MLPrior, we compared it with multiple test prioritization approaches. The considered methods include DeepGini (ISSTA 2020) [3], VanillaSM (ISSTA 2022) [7], Prediction-Confidence Score (ISSTA 2022) [7], and Entropy (ISSTA 2022) [7]. We select these comparative methods because 1) they can be adapted to classical ML models for test prioritization; 2) their effectiveness on DNNs has been demonstrated [7, 3].

- **DeepGini** [3] DeepGini operates by assessing the model's uncertainty in its predictions for tests. The fundamental premise of DeepGini is that tests for which the model exhibits greater uncertainty in its predictions are deemed to have a higher likelihood of being incorrectly predicted. Consequently, these tests will be prioritized higher. The mechanism for calculating this uncertainty in DeepGini is encapsulated in a specific formula, referred to as Formula 5.1. In this formula, the symbol $\xi(t)$ denotes the model's uncertainty regarding its prediction for a particular test t. The higher the value of $\xi(t)$, the greater the uncertainty associated with the model's prediction for the test t, and t will be prioritized higher. By prioritizing tests with higher values of $\xi(t)$, DeepGini can identify and prioritize test inputs that are potentially misclassified.

$$\xi(t) = 1 - \Sigma_{i=1}^{N} p_{t,i}^2 \tag{4.2}$$

  where N is the number of classes, and $p_{t,i}$ denotes the probability of the model predicting t belonging to class i.
- **VanillaSM** [7] The VanillaSM algorithm ranks all the tests by computing the difference between the highest activation probability within the output softmax layer for each test and 1. The calculation is defined by Formula 5.4. A lower value of $V(t)$ indicates that the test is more likely to be misclassified by the model.

$$V(t) = 1 - \max_{i=1}^{N} l_i(t) \tag{4.3}$$

  where N is the number of classes. $\max_{i=1}^{N} l_i(t)$ represents the model's prediction probability for the most confident classification of test t among all N classes.
- **Prediction-Confidence Score (PCS)** PCS [7] prioritizes test inputs by calculating the difference between the probabilities of the model's most confident class and the second most confident class for each test. The formula is given as Formula 5.2. A smaller PCS(t) indicates that a test is more likely to be mispredicted by the model.

$$PCS(t) = p^1(t) - p^2(t) \tag{4.4}$$

  where $p^1(t)$ is the predicted probability of the model for the most confident class of test t, and $p^2(t)$ is the predicted probability of the model for the second most confident class of test t.
- **Entropy** Entropy [7] ranks all tests by calculating the entropy value of the model's predicted probability vector for each test. A higher entropy value for a test indicates that it is more likely to be mispredicted by the model.

- **Random selection** [133] In random selection, the order of test input execution is determined randomly.

4.4.4 Measurements

Following the existing work [3], we employed two metrics to evaluate the effectiveness of MLPrior, the compared approaches, and the variants of MLPrior: Average Percentage of Fault Detection (APFD) [92] and Percentage of Faults Detected (PFD) [3].

- **Average Percentage of Fault-Detection (APFD)** APFD is a well-established metric utilized for evaluating the effectiveness of test prioritization. A higher APFD value indicates greater effectiveness. The APFD values are computed using Formula 5.5.

$$APFD = 1 - \frac{\sum_{i=1}^{k} o_i}{kn} + \frac{1}{2n} \tag{4.5}$$

  where n denotes the number of test inputs in the test set, and k represents the number of misclassified inputs. o_i is the index of the i_{th} misclassified test within the prioritized test set. Below, we explain from a formula perspective why larger APFD values indicate high test prioritization effectiveness.
  Firstly, in the formula, since n is a constant, a larger APFD value means that the value of $\sum_{i=1}^{k} o_i$ (i.e., the total index sum of misclassified tests within the prioritized list) is smaller. A smaller $\sum_{i=1}^{k} o_i$ implies that the misclassified tests are relatively positioned toward the front of the prioritized test set. This indicates that the misclassified tests are indeed prioritized at the beginning of the test set through the test prioritization approach, thus demonstrating that its effectiveness is high. Following prior work [3], we normalize the APFD values to [0,1]. A prioritization approach is considered better when the APFD value is closer to 1.
- **Percentage of Fault Detected (PFD)** PFD quantifies the ratio of detected misclassified test inputs to the total number of misclassified tests. A higher PFD value suggests that a test prioritization approach is more effective. The calculation of PFD follows Formula 7.6.

$$PFD = \frac{\#F^d}{\#F} \tag{4.6}$$

  where $\#F^d$ is the number of detected misclassified test inputs. $\#F$ is the total number of misclassified test inputs. In our study, we measured the PFD values of MLPrior and compared test prioritization approaches using varying ratios of prioritized tests.

4.4.5 Implementation and Configuration

In terms of the compared approaches, we employed the available implementations provided by their respective authors [7, 3]. Concerning the XGBoost ranking model, we utilized XGBoost version 1.4.2 [8]. For the ranking models Decision Tree, KNN, Logistic Regression, and GaussianNB, we utilized the package provided by scikit-learn 0.24.2 [134]. Regarding the parameters of the ranking models, we set the *n_estimators* parameter of XGBoost to 100. We set the *max_iter* parameter of Logistic Regression to 100. For the Decision Tree ranking algorithm, we set the *min_samples_split* parameter to 2. The *var_smoothing* parameter of GaussianNB was set to 1e-9. Additionally, we set the *n_neighbors* parameter of KNN to 5.

Table 4.2: Effectiveness comparison among MLPrior and DNN test prioritization approaches in terms of APFD on natural datasets (Binary Classification)

Approach	Adult					Bank					Stroke				
	Tree	KNN	LR	NB	XGB	Tree	KNN	LR	NB	XGB	Tree	KNN	LR	NB	XGB
Random	0.508	0.506	0.493	0.505	0.500	0.502	0.494	0.504	0.490	0.494	0.519	0.505	0.502	0.497	0.499
DeepGini	0.739	0.738	0.688	0.710	0.793	0.770	0.704	0.769	0.694	0.837	0.768	0.604	0.593	0.615	0.758
Entropy	0.739	0.738	0.688	0.710	0.793	0.770	0.704	0.769	0.694	0.837	0.768	0.604	0.593	0.615	0.758
PCS	0.739	0.738	0.688	0.710	0.793	0.770	0.704	0.769	0.694	0.837	0.768	0.604	0.593	0.615	0.758
VanillaSM	0.739	0.738	0.688	0.710	0.793	0.770	0.704	0.769	0.694	0.837	0.768	0.604	0.593	0.615	0.758
MLPrior	0.810	0.811	0.829	0.830	0.813	0.863	0.872	0.878	0.877	0.868	0.990	0.787	0.845	0.839	0.900

Furthermore, concerning model mutation, we generated 100 mutant models for each original classical ML model. For dataset mutation, we generated 20 mutant datasets for each natural dataset. In other words, MLPrior generates 20 mutated inputs for each test. Moreover, we conducted a statistical analysis to mitigate the impact of randomness. For each subject (i.e., a dataset with a model), we repeated the experiments 5 times and reported the average results. We conducted the experiments on a high-performance cluster, and each cluster node runs a 2.6 GHz Intel Xeon Gold 6132 CPU with an NVIDIA Tesla V100 16G SXM2 GPU. In terms of data processing, we conducted corresponding experiments on a MacBook Pro laptop with Mac OS Big Sur 11.6, Intel Core i9 CPU, and 64 GB RAM.

4.5 Study Results

4.5.1 RQ1: Effectiveness and Efficiency of MLPrior

Objectives: We evaluate the effectiveness and efficiency of MLPrior in prioritizing test inputs for classical ML models.

Experimental design: We conducted experiments to evaluate the performance of MLPrior from three perspectives:

- **Effectiveness** To assess the effectiveness of MLPrior, we carefully designed 15 subjects consisting of three prevalent datasets, each paired with five classical ML models. Detailed information regarding the subjects can be found in Table 4.1. Moreover, we compared MLPrior against a range of DNN prioritization approaches, namely DeepGini [3], Vanilla Softmax [7], Prediction-Confidence Score (PCS) [7], Entropy [7], and Random Selection. To measure the effectiveness, we used the APFD metric [92] and the PFD metric [3], which are widely-adopted measures for evaluating test prioritization techniques.
- **Efficiency** We evaluate the efficiency of MLPrior by quantifying the time required for each step of MLPrior, as well as the time cost of each compared approach.
- **Statistical analysis** Considering the randomness associated with the training process of the ML models and the MLPrior approach, we conduct a statistical analysis to ensure the stability of our research. More specifically, we replicated all the experiments a total of five times, calculating average results to report in this section. Furthermore, we calculated the p-values to evaluate the statistical significance of our findings.

Results: The experimental results pertaining to RQ1 are presented in Table 4.2, Table 4.3, Figure 4.3, Table 4.4 and Table 4.5. We highlight the approach with the highest effectiveness in grey to facilitate quick and easy interpretation of the results.

When applied to natural inputs, MLPrior outperforms all the compared methods in terms of APFD across all subjects, with an average improvement of 14.74%∼66.93% over the compared approaches. Table 4.2 exhibits the effectiveness of MLPrior in comparison to the compared test prioriti-

Table 4.3: Effectiveness comparison among MLPrior and DNN test prioritization approaches in terms of APFD on natural datasets (Multiclass classification)

Approach	Diabetes					Heartbeat				
	Tree	KNN	LR	NB	XGB	Tree	KNN	LR	NB	XGB
Random	0.499	0.501	0.498	0.501	0.501	0.497	0.509	0.505	0.501	0.475
DeepGini	0.701	0.678	0.760	0.685	0.760	0.781	0.851	0.772	0.486	0.839
Entropy	0.697	0.677	0.759	0.685	0.760	0.780	0.851	0.761	0.530	0.837
PCS	0.702	0.679	0.760	0.686	0.760	0.779	0.851	0.779	0.485	0.839
VanillaSM	0.702	0.679	0.760	0.685	0.761	0.781	0.852	0.776	0.486	0.840
MLPrior	0.769	0.772	0.767	0.802	0.765	0.897	0.883	0.915	0.639	0.914

Table 4.4: Time cost of MLPrior and the compared test prioritization approaches

Time cost	Approach					
	MLPrior	Random	DeepGini	VanillaSM	PCS	Entropy
Feature generation	3 s	-	-	-	-	-
Ranking model training	15 s	-	-	-	-	-
Prediction	55.133 ms	12.566 ms	1.323 ms	1.020 ms	1.355 ms	114.483 ms

Table 4.5: Effectiveness improvement of MLPrior over the compared approaches in terms APFD on natural datasets

Data Type	Approach	# Best cases	Average APFD	Improvement(%)
Binary Classification	Random	0	0.501	70.46
	DeepGini	0	0.719	18.78
	Entropy	0	0.719	18.78
	PCS	0	0.719	18.78
	VanillaSM	0	0.719	18.78
	MLPrior	15	0.854	-
Multiclass Classification	Random	0	0.498	63.05
	DeepGini	0	0.731	11.08
	Entropy	0	0.734	10.63
	PCS	0	0.732	10.93
	VanillaSM	0	0.732	10.93
	MLPrior	10	0.812	-
ALL	Random	0	0.499	66.93
	DeepGini	0	0.725	14.89
	Entropy	0	0.726	14.74
	PCS	0	0.725	14.89
	VanillaSM	0	0.725	14.89
	MLPrior	25	0.833	-

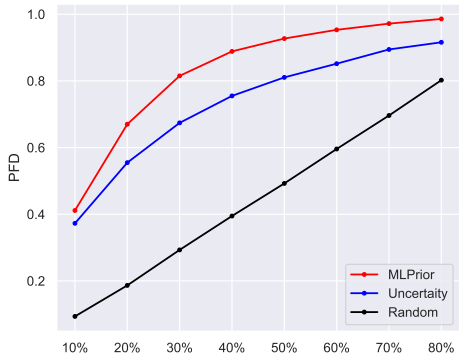


Figure 4.3: Test prioritization effectiveness among MLPrior and the compared approaches (dataset Bank with model GaussianNB). X-Axis: the percentage of prioritized tests; Y-Axis: the percentage of detected miscalssified tests

zation approaches across different subjects. From the table, we see that MLPrior outperforms all the compared methods across all subjects. Specifically, the APFD values of MLPrior range from 0.787 to 0.990, while that of the compared approaches span from 0.494 to 0.837. Table 4.3 demonstrates the effectiveness of MLPrior and the comparative test prioritization methods on multiclass classification datasets. We see that in all cases, the effectiveness of MLPrior is higher than all the comparative methods. Specifically, the APFD range of MLPrior is from 0.639 to 0.915, while the APFD range for the comparative methods is from 0.475 to 0.852. The experimental results demonstrate that MLPrior's effectiveness surpasses all comparative methods on multiclass datasets.

Table 4.5 shows the comparison of effectiveness between MLPrior and other test prioritization methods on all subjects in both binary and multi-class datasets. The evaluation metrics include the number of cases where each method performs the best (denoted as **#Best cases**), the average APFD value of each test prioritization approach (denoted as **Average APFD**), and the improvement of MLPrior relative to each comparison method (denoted as **Improvement(%)**). From Table 4.5, we can see that MLPrior performs the best across all cases, whether in binary or multi-class datasets. In binary datasets, the average APFD of MLPrior is 0.854. In multi-class datasets, it is 0.812, and across all subjects (including both binary and multi-class), it is 0.833. The average APFD of comparison methods across all subjects ranges from 0.499 to 0.726.

Moreover, under all subjects, the average improvement of MLPrior relative to all the compared test prioritization methods ranges from 14.74% to 66.93%. More specifically, in binary datasets, the improvement range of MLPrior relative to all comparison methods is from 18.78% to 70.46%. In multi-class datasets, the improvement range is from 10.93% to 63.05%. These experimental results demonstrate that MLPrior's effectiveness surpasses all other test prioritization methods on natural test inputs.

Figure 4.3 provides a visual comparison between MLPrior and other test prioritization approaches in terms of PFD on the Bank dataset with the GaussianNB model. In this figure, the effectiveness of MLPrior is represented by the red curve, while the blue curve represents the effectiveness of confidence-based test prioritization methods. Additionally, the black curve depicts the baseline effectiveness. It is noteworthy to mention that all confidence-based approaches are consolidated into a single line due to their identical effectiveness across all cases, as evidenced in Table 4.2.

The reason why all confidence-based methods yield the same experimental results on binary classification ML models is as follows: Given a binary classification model, suppose the probability of a test t belonging to category 1 is p, then the probability of it belonging to the other category is $1-p$. Regardless of the confidence-based method used, tests with p values close to 0.5 are deemed more uncertain [3] and thus are prioritized to the front. Therefore, the experimental results of all test prioritization methods are the same. We explain this in detail below.

Feng *et al.* [3] demonstrated that in a binary classification model, if the model's prediction probability for a test is $(0.5, 0.5)$, it means the model is most uncertain about this test, indicating this test is more likely to be misclassified. The closer a test's p value is to 0.5, the more uncertain the model is about that particular test. Consequently, uncertainty is solely determined by p. Regardless of the specific confidence-based test prioritization method employed, tests with p values closer to

0.5 will be prioritized over others.

To illustrate this point, consider a test set with three tests, and the model's probability vectors for these tests are as follows: t_1 $(0.9, 0.1)$, t_2 $(0.7, 0.3)$, t_3 $(0.8, 0.2)$. Irrespective of the chosen confidence-based test prioritization method, the resulting ranking will be $t_2 \rightarrow t_3 \rightarrow t_1$ because t_2 has the p value (0.7) closest to 0.5, followed by t_3 (p = 0.8), while t_1 has the farthest p value from 0.5 (p = 0.9).

From Figure 4.3, we see that MLPrior consistently outperforms all the compared methods across different prioritization ratios. These experimental results strongly suggest that MLPrior exhibits higher effectiveness than other test prioritization approaches in classical ML test prioritization. As stated in the experimental design, due to the inherent randomness associated with the training process, we conducted a statistical analysis. This analysis involved repeating all experiments a total of five times. The p-value of the experimental results was found to be significantly less than 10^{-06}, which suggests that MLPrior can stably outperform the compared test prioritization approaches.

MLPrior showcases acceptable efficiency, with an average execution time of less than 20 seconds. In addition to evaluating its effectiveness, we also compared the efficiency of MLPrior with other test prioritization approaches, and the experimental results are presented in Table 4.4. The findings indicate that the average total running time of MLPrior on each subject is under 20 seconds, which can be broken down into three main components: feature generation (3 seconds), ranking model training (15 seconds), and prediction (55.133 ms). Here, 'ms' refers to milliseconds. The prediction times for the confidence-based test prioritization methods are as follows: DeepGini: 1.323 ms; VanillasM: 1.020 ms; PCS: 1.355 ms; Entropy: 114.483 ms. While confidence-based test prioritization techniques exhibit higher efficiency with a running time of less than 1 second, the computational cost of MLPrior remains reasonable in practical scenarios, especially considering the laborious and costly nature of manual labeling. Despite being slightly less efficient than confidence-based methods, the considerable improvement in effectiveness demonstrated by MLPrior, ranging from 18.78% to 70.46% compared to those techniques, underscores its overall performance.

Answer to RQ1: *When applied to natural inputs, MLPrior outperforms all the compared methods in terms of APFD across all subjects, with an average improvement of 14.74%∼66.93% over the compared approaches. Moreover, MLPrior showcases acceptable efficiency, with an average execution time of less than 20 seconds.*

4.5.2 RQ2: Effectiveness of MLPrior on different types of test inputs

Objectives: In addition to assessing MLPrior's performance on natural test sets, we also evaluate its effectiveness on different types of test inputs, encompassing mixed noisy data and fairness data. *Mixed noisy datasets* are composed of 70% natural data and 30% of noisy data. *Fairness datasets* are constructed with the aim of avoiding biases associated with individual attributes, such as gender and age. Ensuring fairness in machine learning is crucial to prevent bias and discrimination against specific groups during predictions. Fairness has emerged as a critical ethical consideration across diverse machine learning domains, such as recruitment, loan

Table 4.6: Effectiveness comparison among MLPrior and DNN test prioritizatiion approaches in terms of APFD on mixed noisy datasets (Binary Classification)

Approach	Adult					Bank					Stroke				
	Tree	KNN	LR	NB	XGB	Tree	KNN	LR	NB	XGB	Tree	KNN	LR	NB	XGB
Random	0.499	0.499	0.500	0.500	0.501	0.498	0.502	0.497	0.502	0.500	0.509	0.500	0.499	0.501	0.501
DeepGini	0.460	0.701	0.682	0.711	0.744	0.740	0.707	0.766	0.676	0.826	0.702	0.602	0.593	0.611	0.755
Entropy	0.460	0.701	0.682	0.711	0.744	0.740	0.707	0.766	0.676	0.826	0.702	0.602	0.593	0.611	0.755
PCS	0.460	0.701	0.682	0.711	0.744	0.740	0.707	0.766	0.676	0.826	0.702	0.602	0.593	0.611	0.755
VanillaSM	0.460	0.701	0.682	0.711	0.744	0.740	0.707	0.766	0.676	0.826	0.702	0.602	0.593	0.611	0.755
MLPrior	0.830	0.810	0.825	0.827	0.829	0.867	0.872	0.875	0.875	0.868	0.982	0.825	0.845	0.838	0.898

Table 4.7: Effectiveness comparison among MLPrior and DNN test prioritizatiion approaches in terms of APFD on mixed noisy datasets (Multiclass classification)

Approach	Diabetes					Heartbeat				
	Tree	KNN	LR	NB	XGB	Tree	KNN	LR	NB	XGB
Random	0.501	0.499	0.501	0.501	0.501	0.502	0.500	0.499	0.500	0.493
DeepGini	0.670	0.658	0.751	0.707	0.725	0.773	0.851	0.772	0.486	0.839
Entropy	0.671	0.658	0.749	0.705	0.725	0.771	0.851	0.761	0.530	0.837
PCS	0.671	0.661	0.751	0.710	0.726	0.771	0.851	0.779	0.485	0.839
VanillaSM	0.670	0.659	0.751	0.707	0.725	0.772	0.851	0.776	0.486	0.841
MLPrior	0.789	0.776	0.772	0.801	0.773	0.901	0.878	0.916	0.639	0.908

approvals, and medical diagnosis [65]. In these domains, the absence of fairness can result in unjust treatment of certain groups, significantly impacting individuals' lives and rights.

Our investigation revolves around two primary sub-questions:

- **RQ-2.1** How does MLPrior perform on mixed noisy data?
- **RQ-2.2** How does MLPrior perform on fairness data?

Experimental design: We conduct the following experiments to answer the aforementioned sub-questions.

[**Experiment ❶**] In the first step, we generate noisy data from the three natural datasets used in RQ1 (i.e., Adult, Bank, and Stroke). To this end, we mix 30% noisy data with 70% natural data to create mixed noisy data. The reason we chose a noise generation ratio of 30% is as follows: A high noise ratio, such as 90%, would result in a significant proportion of noisy test inputs, and a substantial number of misclassified tests would be selected by any prioritization method, thereby complicating the demonstration of MLPrior's effectiveness. Therefore, in order to ensure an efficacious evaluation of both MLPrior and the comparative approaches, we opted for a reasonable noise generation ratio (i.e., 30%). For each of the three natural datasets, we generate ten mixed noisy datasets, resulting in 30 (3×10) mixed datasets. Each mixed dataset is paired with five classical ML models, leading to a total of 150 subjects (30 datasets $\times$ 5 models). Based on these generated subjects, we compare the effectiveness of MLPrior with other test prioritization methods.

[**Experiment ❷**] To generate fairness data for evaluation, we adopt the approach used in previous research [66]. Specifically, for each natural dataset utilized in RQ1 (i.e., Adult, Bank, and Stroke), we randomly selected a subset of instances from the original test set and modified their gender and age attribute values while keeping the original labels untouched. The reason for ensuring the labels untouched is as follows: In the context of ensuring fairness, the model should maintain consistent classification results when the protected attributes (such as genders and ages) are changed, while all other attributes remain unaltered.

Concretely, for the attribute "gender", we changed half of the "male" to "females" and half of the "females" to "males". Regarding the attribute "age", following the prior work [119], we modified the "middle age" (30∼59) instances in the test set to

Table 4.8: Effectiveness improvement of MLPrior over the compared approaches in terms of APFD on mixed noisy datasets

Data Type	Approach	# Best cases	Average APFD	Improvement(%)
Binary Classification	Random	0	0.499	63.32
	DeepGini	0	0.685	25.26
	Entropy	0	0.685	25.26
	PCS	0	0.685	25.26
	VanillaSM	0	0.685	25.26
	MLPrior	150	0.858	-
Multiclass Classification	Random	0	0.499	63.32
	DeepGini	0	0.723	12.72
	Entropy	0	0.726	12.25
	PCS	0	0.724	12.57
	VanillaSM	0	0.723	12.72
	MLPrior	100	0.815	-
ALL	Random	0	0.499	67.73
	DeepGini	0	0.704	18.89
	Entropy	0	0.706	18.55
	PCS	0	0.705	18.72
	VanillaSM	0	0.704	18.89
	MLPrior	250	0.837	-

Table 4.9: Effectiveness comparison among MLPrior and DNN test prioritization approaches in terms of APFD on fairness datasets (Binary Classification)

Approach	Age Change					Gender Change				
	Tree	KNN	LR	NB	XGB	Tree	KNN	LR	NB	XGB
Random	0.504	0.493	0.500	0.499	0.503	0.484	0.499	0.496	0.503	0.499
DeepGini	0.670	0.704	0.726	0.692	0.788	0.730	0.668	0.636	0.657	0.774
Entropy	0.670	0.704	0.726	0.692	0.788	0.730	0.668	0.636	0.657	0.774
PCS	0.670	0.704	0.726	0.692	0.788	0.730	0.668	0.636	0.657	0.774
VanillaSM	0.670	0.704	0.726	0.692	0.788	0.730	0.668	0.636	0.657	0.774
MLPrior	0.847	0.843	0.852	0.852	0.842	0.897	0.813	0.834	0.834	0.856

"young age" (18∼29) while converting the "young age" test instances to "middle age." Using the generated fairness test sets, we compare the effectiveness of MLPrior with other test prioritization methods.

Results: The experimental findings pertaining to RQ2.1 are presented in Table 4.6, Table 4.7, Table 4.8. Table 4.6 showcases the effectiveness difference between MLPrior and the compared test prioritization methods when applied to mixed noisy inputs. The evaluation metric employed is the Average Percentage of Faults Detected (APFD). We highlight the approach with the highest effectiveness in grey to facilitate easy interpretation of the results.

On mixed noisy inputs, MLPrior consistently performs better than all the compared approaches, with an average improvement of 18.55%∼67.73%. From Table 4.6, we see that MLPrior consistently outperforms all the compared methods across each case. Remarkably, the APFD values achieved by MLPrior range from 0.810 to 0.982, while that of the compared methods range from 0.497 to 0.766.

Table 4.10: Effectiveness comparison among MLPrior and DNN test prioritization approaches in terms of APFD on fairness datasets (Multiclass classification)

Approach	Age Change					Gender Change				
	Tree	KNN	LR	NB	XGB	Tree	KNN	LR	NB	XGB
Random	0.496	0.499	0.495	0.496	0.498	0.502	0.498	0.503	0.501	0.497
DeepGini	0.652	0.660	0.730	0.691	0.717	0.697	0.669	0.757	0.684	0.759
Entropy	0.649	0.660	0.729	0.690	0.717	0.694	0.669	0.757	0.683	0.759
PCS	0.653	0.662	0.730	0.692	0.717	0.697	0.671	0.758	0.684	0.759
VanillaSM	0.652	0.661	0.730	0.691	0.717	0.698	0.670	0.758	0.683	0.759
MLPrior	0.776	0.771	0.767	0.798	0.765	0.773	0.773	0.767	0.801	0.765

Table 4.11: Effectiveness improvement of MLPrior over the compared approaches in terms of APFD on fairness datasets

Data Type	Approach	# Best cases	Average APFD	Improvement(%)
Binary Classification	Random	0	0.498	70.08
	DeepGini	0	0.705	20.14
	Entropy	0	0.705	20.14
	PCS	0	0.705	20.14
	VanillaSM	0	0.705	20.14
	MLPrior	20	0.847	-
Multiclass Classification	Random	0	0.499	61.69
	DeepGini	0	0.702	10.54
	Entropy	0	0.701	10.70
	PCS	0	0.702	10.54
	VanillaSM	0	0.702	10.54
	MLPrior	10	0.776	-
ALL	Random	0	0.499	62.72
	DeepGini	0	0.704	15.34
	Entropy	0	0.703	15.50
	PCS	0	0.704	15.34
	VanillaSM	0	0.704	15.34
	MLPrior	30	0.812	-

Table 4.7 presents the effectiveness of MLPrior compared to other test prioritization methods on noisy datasets for multiclassification. We see that MLPrior outperforms all other test prioritization methods across all multiclassification subjects. The range of APFD values for MLPrior is from 0.639 to 0.916, whereas the range for the compared test prioritization methods is from 0.485 to 0.851. We conclude that on noisy datasets for multiclassification, the effectiveness of MLPrior surpasses that of the compared test prioritization methods.

Table 4.8 provides an overall comparison of the effectiveness of MLPrior and other test prioritization methods on binary classification datasets, multiclass classification datasets, and all subjects (both binary and multiclass). The evaluation metrics include the number of cases where each method performs the best (denoted as **#Best cases**), the average APFD value of each test prioritization approach (denoted as **Average APFD**), and the improvement of MLPrior relative to each comparison method (denoted as **Improvement(%)**).

In Table 4.8, we observe that MLPrior performs the best across all subjects, regardless of whether they are binary or multiclass. The average APFD of MLPrior on all subjects (including both binary and multiclass) is 0.837. Specifically, the average APFD of MLPrior in binary classification is 0.858, while in multiclass classification, it is 0.815. In contrast, the range of the average APFD for the comparison methods across all subjects is from 0.499 to 0.706. Moreover, across all subjects, the average improvement of MLPrior relative to the comparison test prioritization methods ranges from 18.55% to 67.73%.

Answer to RQ2.1: *On mixed noisy inputs, MLPrior consistently performs better than all the compared approaches, with an average improvement of 18.55%∼67.73%.*

The experimental results of RQ2.2 are presented in Table 4.9, Table 4.10, Table 4.11. Table 4.9 displays the effectiveness differences between MLPrior and all the comparative methods on the fairness dataset in terms of APFD. The gray shading indicates the best-performing method for each case.

On fairness data, MLPrior consistently performs better than all the compared approaches, with an average improvement of 15.34%∼62.72%.

We see that MLPrior achieves the highest effectiveness across all cases, with an APFD range of 0.813 to 0.897. In contrast, the comparative methods have an APFD range of 0.484 to 0.788.

Table 4.10 showcases the effectiveness of MLPrior compared to other test prioritization methods on fairness datasets for multiclassification. We can see that MLPrior exceeds the performance of all other test prioritization methods in all multiclassification subjects. The APFD values for MLPrior range from 0.765 to 0.801, while the compared test prioritization methods range between 0.495 and 0.759. The experimental results demonstrate that, in the context of fairness datasets for multiclassification, MLPrior's effectiveness is superior to that of the other compared test prioritization methods.

Table 4.11 presents a comparative analysis of the effectiveness between MLPrior and other test prioritization methods across all fairness subjects within binary and multi-class datasets. The evaluation metrics encompass the number of instances where each method is most effective (denoted as **#Best cases**), the average APFD value for each test prioritization approach (denoted as **Average APFD**), and the relative improvement of MLPrior compared to each method (denoted as **Improvement(%)**). According to Table 4.11, MLPrior consistently outperforms other methods in all scenarios, whether in binary or multi-class datasets. Specifically, in binary datasets, MLPrior's average APFD is 0.847. In multi-class datasets, it is 0.776, and the overall average across all subjects (encompassing both binary and multi-class) stands at 0.812. The average APFD for the comparison methods across all subjects varies from 0.499 to 0.704.

Furthermore, across all fairness subjects, the average improvement of MLPrior compared to all other test prioritization methods ranges from 15.34% to 62.72%. More specifically, within binary datasets, MLPrior's improvement over the comparison methods varies from 20.14% to 70.08%. In multi-class datasets, this improvement range is between 10.54% and 61.69%. These experimental results indicate that MLPrior's effectiveness is superior to all other test prioritization methods when dealing with fairness test inputs.

Answer to RQ2.2: *On fairness data, MLPrior consistently performs better than all the compared approaches, with an average improvement of 15.34%∼62.72%*

4.5.3 RQ3: Impact of ranking models on the effectiveness of MLPrior

Objectives: We investigate the impact of different ranking models on the effectiveness of MLPrior.

Experimental design: In order to investigate the impact of different ranking models, we propose four variants of MLPrior denoted as MLPrior^T, MLPrior^K, MLPrior^L, and MLPrior^N. These variants employ the ranking models decision tree [112], K-nearest neighbors (KNN)[69], logistic regression[14], and Gaussian Naive Bayes (GaussianNB) [113], respectively. The only difference between them and the original MLPrior lies in the selection of the ranking models, while the rest of the workflow remains unchanged. We utilize the APFD metric to evaluate the effectiveness differences of MLPrior, these variants, and the comparative test prioritization methods on natural and mixed noisy datasets.

Results: The experimental results for RQ3 are presented in Table 4.12, Table 4.13,

Table 4.12: Effectiveness comparison among MLPrior, MLPrior Variants and DNN test prioritization approaches in terms of APFD on natural datasets (Binary Classification)

Approach	Adult					Bank					Stroke				
	Tree	KNN	LR	NB	XGB	Tree	KNN	LR	NB	XGB	Tree	KNN	LR	NB	XGB
Random	0.508	0.506	0.493	0.505	0.500	0.502	0.494	0.504	0.490	0.494	0.519	0.505	0.502	0.497	0.499
DeepGini	0.739	0.738	0.688	0.710	0.793	0.770	0.704	0.769	0.694	0.837	0.768	0.604	0.593	0.615	0.758
Entropy	0.739	0.738	0.688	0.710	0.793	0.770	0.704	0.769	0.694	0.837	0.768	0.604	0.593	0.615	0.758
PCS	0.739	0.738	0.688	0.710	0.793	0.770	0.704	0.769	0.694	0.837	0.768	0.604	0.593	0.615	0.758
VanillaSM	0.739	0.738	0.688	0.710	0.793	0.770	0.704	0.769	0.694	0.837	0.768	0.604	0.593	0.615	0.758
MLPrior^{T}	0.706	0.715	0.786	0.787	0.742	0.747	0.790	0.801	0.817	0.779	0.839	0.753	0.837	0.832	0.889
MLPrior^{K}	0.796	0.784	0.746	0.749	0.738	0.833	0.786	0.795	0.796	0.803	0.774	0.635	0.604	0.617	0.679
MLPrior^{L}	0.740	0.743	0.722	0.672	0.688	0.786	0.769	0.757	0.771	0.751	0.898	0.621	0.608	0.608	0.703
MLPrior^{N}	0.787	0.775	0.737	0.739	0.729	0.823	0.775	0.784	0.782	0.792	0.765	0.626	0.589	0.604	0.669
MLPrior	0.810	0.811	0.829	0.830	0.813	0.863	0.872	0.878	0.877	0.868	0.990	0.787	0.845	0.839	0.900

Table 4.13: Effectiveness comparison among MLPrior, MLPrior Variants and DNN test prioritization approaches in terms of APFD on natural datasets (Multiclass classification)

Approach	Diabetes					Heartbeat				
	Tree	KNN	LR	NB	XGB	Tree	KNN	LR	NB	XGB
Random	0.499	0.501	0.498	0.501	0.501	0.497	0.509	0.505	0.501	0.475
DeepGini	0.701	0.678	0.760	0.685	0.760	0.781	0.851	0.772	0.486	0.839
Entropy	0.697	0.677	0.759	0.685	0.760	0.780	0.851	0.761	0.530	0.837
PCS	0.702	0.679	0.760	0.686	0.760	0.779	0.851	0.779	0.485	0.839
VanillaSM	0.702	0.679	0.760	0.685	0.761	0.781	0.851	0.776	0.486	0.840
MLPrior^{T}	0.667	0.686	0.695	0.765	0.691	0.732	0.699	0.847	0.634	0.737
MLPrior^{K}	0.649	0.654	0.666	0.756	0.654	0.799	0.727	0.890	0.638	0.815
MLPrior^{L}	0.766	0.769	0.762	0.787	0.759	0.792	0.750	0.804	0.625	0.743
MLPrior^{N}	0.752	0.756	0.741	0.774	0.730	0.746	0.689	0.707	0.580	0.673
MLPrior	0.769	0.772	0.767	0.802	0.765	0.897	0.883	0.915	0.639	0.914

Table 4.14: Effectiveness comparison among MLPrior, MLPrior Variants, and DNN test prioritization approaches in terms of APFD on mixed noisy datasets (Binary Classification)

Approach	Adult					Bank					Stroke				
	Tree	KNN	LR	NB	XGB	Tree	KNN	LR	NB	XGB	Tree	KNN	LR	NB	XGB
Random	0.499	0.499	0.500	0.500	0.501	0.498	0.502	0.497	0.502	0.500	0.509	0.500	0.499	0.501	0.501
DeepGini	0.460	0.701	0.682	0.711	0.744	0.740	0.707	0.766	0.676	0.826	0.702	0.602	0.593	0.611	0.755
Entropy	0.460	0.701	0.682	0.711	0.744	0.740	0.707	0.766	0.676	0.826	0.702	0.602	0.593	0.611	0.755
PCS	0.460	0.701	0.682	0.711	0.744	0.740	0.707	0.766	0.676	0.826	0.702	0.602	0.593	0.611	0.755
VanillaSM	0.460	0.701	0.682	0.711	0.744	0.740	0.707	0.766	0.676	0.826	0.702	0.602	0.593	0.611	0.755
MLPrior^{T}	0.771	0.740	0.776	0.781	0.773	0.812	0.791	0.801	0.815	0.780	0.848	0.784	0.836	0.831	0.887
MLPrior^{K}	0.751	0.753	0.728	0.736	0.745	0.821	0.793	0.787	0.792	0.795	0.736	0.633	0.602	0.603	0.673
MLPrior^{L}	0.680	0.714	0.678	0.677	0.688	0.813	0.745	0.760	0.766	0.759	0.830	0.626	0.607	0.605	0.690
MLPrior^{N}	0.741	0.746	0.718	0.727	0.736	0.813	0.783	0.778	0.783	0.790	0.727	0.623	0.592	0.593	0.662
MLPrior	0.830	0.810	0.825	0.827	0.829	0.867	0.872	0.875	0.875	0.868	0.982	0.825	0.845	0.838	0.898

Table 4.15: Effectiveness comparison among MLPrior, MLPrior Variants and DNN test prioritization approaches in terms of APFD on mixed noisy datasets (Multiclass classification)

Approach	Diabetes					Heartbeat				
	Tree	KNN	LR	NB	XGB	Tree	KNN	LR	NB	XGB
Random	0.501	0.499	0.501	0.501	0.501	0.502	0.500	0.499	0.500	0.493
DeepGini	0.670	0.658	0.751	0.707	0.725	0.773	0.851	0.772	0.486	0.839
Entropy	0.671	0.658	0.749	0.705	0.725	0.771	0.851	0.761	0.53	0.837
PCS	0.671	0.661	0.751	0.710	0.726	0.771	0.851	0.779	0.485	0.839
VanillaSM	0.671	0.659	0.751	0.707	0.725	0.772	0.851	0.776	0.486	0.841
MLPrior^T	0.730	0.698	0.699	0.761	0.701	0.733	0.681	0.853	0.634	0.744
MLPrior^K	0.762	0.751	0.745	0.769	0.735	0.755	0.683	0.709	0.582	0.660
MLPrior^L	0.776	0.764	0.76	0.783	0.760	0.794	0.742	0.808	0.625	0.736
MLPrior^N	0.762	0.751	0.745	0.769	0.735	0.755	0.683	0.709	0.582	0.660
MLPrior	0.789	0.776	0.772	0.801	0.773	0.901	0.878	0.916	0.639	0.908

Table 4.16: Effectiveness comparison among MLPrior, MLPrior Variants and DNN test prioritization approaches in terms of APFD on fairness datasets (Binary Classification & Multiclass classification)

Data Type	Approach	Age Change					Gender Change				
		Tree	KNN	LR	NB	XGB	Tree	KNN	LR	NB	XGB
Binary Classification	Random	0.504	0.493	0.500	0.499	0.503	0.484	0.499	0.496	0.503	0.499
	DeepGini	0.670	0.704	0.726	0.692	0.788	0.730	0.668	0.636	0.657	0.774
	Entropy	0.670	0.704	0.726	0.692	0.788	0.730	0.668	0.636	0.657	0.774
	PCS	0.670	0.704	0.726	0.692	0.788	0.730	0.668	0.636	0.657	0.774
	VanillaSM	0.670	0.704	0.726	0.692	0.788	0.730	0.668	0.636	0.657	0.774
	MLPrior^T	0.782	0.761	0.790	0.805	0.761	0.766	0.748	0.806	0.809	0.813
	MLPrior^K	0.722	0.727	0.735	0.737	0.763	0.655	0.655	0.711	0.712	0.705
	MLPrior^L	0.773	0.737	0.733	0.729	0.718	0.821	0.682	0.659	0.648	0.687
	MLPrior^N	0.786	0.775	0.752	0.765	0.763	0.782	0.699	0.657	0.670	0.705
	MLPrior	0.847	0.843	0.852	0.852	0.842	0.897	0.813	0.834	0.834	0.856
Multiclass Classification	Random	0.496	0.499	0.495	0.496	0.498	0.502	0.498	0.503	0.501	0.497
	DeepGini	0.652	0.660	0.730	0.691	0.717	0.697	0.669	0.757	0.684	0.759
	Entropy	0.649	0.660	0.729	0.690	0.717	0.694	0.669	0.757	0.683	0.759
	PCS	0.653	0.662	0.730	0.692	0.717	0.697	0.671	0.758	0.684	0.759
	VanillaSM	0.652	0.661	0.730	0.691	0.717	0.698	0.670	0.758	0.683	0.759
	MLPrior^T	0.705	0.687	0.694	0.757	0.689	0.685	0.695	0.690	0.765	0.689
	MLPrior^K	0.735	0.750	0.735	0.767	0.727	0.754	0.754	0.742	0.776	0.730
	MLPrior^L	0.765	0.763	0.759	0.783	0.757	0.770	0.769	0.760	0.789	0.758
	MLPrior^N	0.735	0.750	0.735	0.767	0.727	0.754	0.754	0.742	0.776	0.730
	MLPrior	0.776	0.771	0.767	0.798	0.765	0.773	0.773	0.767	0.801	0.765

Table 4.17: Effectiveness improvement of MLPrior over MLPrior Variants, and DNN test prioritization approaches

Approach	# Best cases	Average APFD	Improvement(%)
Random	0	0.499	65.73
DeepGini	0	0.711	16.32
Entropy	0	0.712	16.15
PCS	0	0.711	16.32
VanillaSM	0	0.711	16.32
MLPrior^T	0	0.770	7.40
MLPrior^K	0	0.697	18.65
MLPrior^L	0	0.719	15.02
MLPrior^N	0	0.683	21.08
MLPrior	305	0.827	-

Table 4.14, Table 4.15, Table 4.16 and Table 4.17. Tables 4.12 and Table 4.13 display the effectiveness of MLPrior, its variants, and the compared test prioritization methods on natural datasets. Tables 4.14 and Table 4.15 show their effectiveness on noisy datasets. Table 4.16 presents their effectiveness on fairness datasets. Table 4.17 illustrates their average performance across all datasets (including natural, noisy, and fairness datasets), as well as the improvements of MLPrior relative to its variants and the compared test prioritization methods.

MLPrior outperforms all its variants in test prioritization, indicating that among all ranking models, the XGBoost model (utilized by the original MLPrior) can better utilize the generated features of test inputs for test prioritization. Table 4.12 and Table 4.13 demonstrate the effectiveness of MLPrior on **natural** datasets, including binary classification datasets (Table 4.12) and multiclass classification datasets (Table 4.13). We see that, whether on binary or multiclass datasets, the effectiveness of MLPrior (measured by APFD) consistently surpasses all its variants. On binary natural datasets (Table 4.12), the APFD range for MLPrior is from 0.8110 to 0.990, while the range for its variants is from 0.589 to 0.898. On multiclass natural datasets (Table 4.13), the APFD range for MLPrior is from 0.639 to 0.915, while the range for its variants is from 0.580 to 0.890. We conclude that on **natural** datasets, the effectiveness of MLPrior exceeds all its variants. Moreover, on **noisy** datasets, encompassing both binary classification datasets (Table 4.14) and multiclass classification datasets (Table 4.15), MLPrior also consistently outperforms all its variants across all cases.

Table 4.16 demonstrates the effectiveness of MLPrior, its variants, and the compared test prioritization methods on **fairness** datasets. We see that, whether on fairness datasets constructed based on age or those constructed based on gender, the effectiveness of MLPrior outperforms both its variants and all test prioritization methods. Specifically, the APFD range for MLPrior is from 0.765 to 0.897, while the range for its variants is from 0.648 to 0.821. We conclude that, on **fairness** datasets, the effectiveness of MLPrior exceeds all its variants.

Table 4.17 displays the effectiveness of MLPrior, its variants, and the compared test prioritization methods across all datasets (i.e., natural, noisy, and fairness datasets). We see that MLPrior performs the best across all 305 cases. Specifically, these 305 cases represent 25 natural subjects + 250 noisy data subjects + 30 fairness data subjects, totaling 305 cases. The detailed origins of these numbers can be referred to in Section 4.4.2. Additionally, across all subjects, the average effectiveness of MLPrior is 0.827, while the average effectiveness of its variants ranges from 0.683 to 0.770. Furthermore, the improvement of MLPrior over its variants in terms of APFD lies between 7.40% and 21.08%.

The experimental results above demonstrate that MLPrior performs better than its variants, indicating that, among all the ranking models evaluated, the **XGBoost** model used in the original MLPrior demonstrates a better capability in utilizing the generated features of test inputs for test prioritization.

Answer to RQ3: *MLPrior outperforms all its variants in test prioritization, indicating that among all ranking models, the XGBoost model (utilized by the original MLPrior) can better utilize the generated features of test inputs for test prioritization.*

4.5.4 RQ4: Feature contribution Analysis

Objectives: We investigate the contributions of three types of features (i.e., model mutation features, input mutation features, and original attribute features) on the effectiveness of MLPrior.

Experimental design: To assess the impact of different feature types on the effectiveness of MLPrior, we adopt the cover metric from the XGBoost algorithm [8] as the measurement tool. Firstly, within the context of each subject, we compute the importance scores for each generated feature. Subsequently, we identify the top N most contributing features. Based on it, we investigate the extent to which each type of feature contributes to the effectiveness of MLPrior. Below, we explain the working principle of the XGBoost cover metric.

The Working Principle of XGBoost Cover Metric: The cover metric in XGBoost quantifies feature importance by evaluating the average coverage of each instance across the leaf nodes in a decision tree. Specifically, the cover metric calculates the frequency at which a specific feature is utilized to partition the data in all trees of the ensemble. The coverage values associated with each feature across all trees are then summed. Subsequently, the resulting coverage value is normalized by the total number of instances, providing the average coverage of each instance by the leaf nodes. The significance of a particular feature is determined based on its derived coverage value, with features exhibiting higher coverage values being assigned greater importance.

Results: Table 4.18 presents the contributions of different feature types to the effectiveness of MLPrior. In this table, we utilize the abbreviations MMF, IMF, and OAF to represent model mutation features, input mutation features, and original attribute features, respectively. The numbers after the feature abbreviations denote the indices of the corresponding features. For instance, *IMF-123* denotes the input mutation feature with index 123. We conducted the feature contribution analysis on both binary classification datasets (Adult, Bank, and Stroke) and multiclass classification datasets (Diabetes and Heartbeat).

All three types of features (i.e., model mutation features, input mutation features, and original attribute features) visibly contribute to the effectiveness of MLPrior. In Table 4.18, we find that in binary classification datasets, for the majority of cases (14 out of 15), all three types of features are present among the top-N most contributing features. For instance, in the dataset Adult with the LR model, IMF features account for 40% of the top 10 critical features, MMF features account for 50%, and OAF features account for 10%. In the case of dataset Bank with the Tree model, IMF features contribute to 20% of the top 10 critical features, MMF features account for 70%, and OAF features account for 10%. Moreover, regarding the multiclass classification datasets, we find that in all cases (10 out of 10), all three types of features are present among the top-N most contributing features. These experimental findings demonstrate that each type of feature makes a visible contribution to the effectiveness of MLPrior.

> **Answer to RQ4:** *All three types of features (i.e., model mutation features, input mutation features, and original attribute features) visibly contribute to the effectiveness of MLPrior.*

Table 4.18: Top-10 most contributing features for each subject

Data	Rank	Tree		KNN		LR		NB		XGB	
		Feature	Value	Feature	Value	Feature	Value	Feature	Value	Feature	Value
Adult	1	IMF-123	1653	IMF-21	1544	IMF-44	2786	IMF-127	3140	IMF-118	2976
	2	MMF-29	1605	OAF-10	1358	IMF-45	2658	IMF-129	2550	IMF-120	2674
	3	IMF-127	1369	OAF-11	1342	MMF-19	2127	IMF-131	1970	OAF-5	1660
	4	OAF-5	1362	OAF-5	1096	MMF-26	1536	OAF-5	1175	IMF-131	1586
	5	OAF-10	1339	OAF-2	630	IMF-49	1429	IMF-126	1093	IMF-126	1460
	6	MMF-67	1217	OAF-9	585	OAF-5	1252	OAF-10	961	OAF-10	1429
	7	MMF-82	1172	IMF-23	576	MMF-31	1124	OAF-11	793	OAF-11	1140
	8	OAF-11	1016	MMF-17	544	IMF-53	1122	MMF-114	773	IMF-128	809
	9	MMF-43	1005	OAF-13	538	MMF-10	1083	IMF-117	759	OAF-13	778
	10	IMF-129	976	MMF-15	498	MMF-16	1061	IMF-115	692	OAF-2	594
Bank	1	IMF-122	2313	IMF-25	1410	IMF-49	1851	IMF-131	1447	IMF-126	1566
	2	MMF-31	1499	OAF-4	1053	MMF-17	1427	MMF-117	1417	OAF-8	1061
	3	MMF-72	1380	IMF-22	1040	MMF-35	1262	IMF-120	1199	IMF-123	1042
	4	MMF-27	1096	OAF-7	710	IMF-46	1096	OAF-4	1148	OAF-4	964
	5	IMF-127	1080	OAF-8	696	OAF-8	971	OAF-7	1094	IMF-134	953
	6	MMF-58	1001	MMF-18	527	MMF-31	933	IMF-135	794	IMF-129	874
	7	MMF-60	989	OAF-13	508	MMF-27	929	IMF-122	730	OAF-7	753
	8	OAF-4	798	OAF-11	495	IMF-42	923	OAF-8	721	IMF-128	718
	9	MMF-86	785	OAF-5	369	IMF-41	828	OAF-11	584	IMF-122	610
	10	MMF-108	784	OAF-6	363	IMF-55	807	OAF-13	556	OAF-6	521
Stroke	1	IMF-110	1526	OAF-4	1006	MMF-28	1534	IMF-124	844	OAF-4	1523
	2	MMF-83	736	OAF-5	758	MMF-18	1080	IMF-115	783	MMF-111	1322
	3	OAF-2	465	OAF-7	706	IMF-38	985	OAF-7	718	IMF-122	807
	4	MMF-35	430	OAF-8	625	MMF-29	952	OAF-4	648	OAF-2	729
	5	MMF-55	424	OAF-1	530	OAF-4	827	OAF-8	645	OAF-3	706
	6	MMF-28	320	IMF-16	507	IMF-42	801	MMF-113	541	OAF-8	482
	7	OAF-5	251	IMF-18	415	OAF-7	728	IMF-127	530	OAF-7	455
	8	IMF-112	241	MMF-12	401	IMF-40	696	OAF-5	472	IMF-123	393
	9	MMF-45	165	OAF-9	394	OAF-5	589	IMF-122	426	OAF-5	363
	10	OAF-8	129	OAF-3	390	OAF-8	579	IMF-114	352	OAF-1	267
Diabetes	1	ORF-2	9302	ORF-2	6300	IMF-51	12816	ORF-2	10771	IMF-124	18667
	2	ORF-0	8395	ORF-10	4102	IMF-46	11041	IMF-134	8308	IMF-133	18659
	3	MMF-41	7238	ORF-0	3372	MMF-40	7758	IMF-126	6392	IMF-138	17907
	4	MMF-106	6264	ORF-13	3211	IMF-47	7049	ORF-10	5044	MMF-120	16122
	5	MMF-54	6124	ORF-3	3147	IMF-48	6358	IMF-139	4788	IMF-128	13657
	6	ORF-10	5533	ORF-14	2329	MMF-28	6334	IMF-135	3858	IMF-102	11819
	7	MMF-71	5349	MMF-21	2257	ORF-0	5680	IMF-132	3677	ORF-2	10277
	8	MMF-22	5333	ORF-18	2118	ORF-2	5679	MMF-121	3668	ORF-10	3819
	9	MMF-44	5157	ORF-11	2100	IMF-44	5375	IMF-125	3413	IMF-140	3684
	10	IMF-125	4837	IMF-28	1991	MMF-27	5206	IMF-140	3309	ORF-0	3600
Heartbeat	1	MMF-211	3488	IMF-210	3627	ORF-155	3345	ORF-121	1127	IMF-310	2589
	2	MMF-277	3007	ORF-171	1923	IMF-236	2690	ORF-124	788	IMF-317	1718
	3	MMF-266	2084	ORF-77	1689	IMF-232	1888	IMF-315	420	IMF-308	1296
	4	IMF-307	1943	MMF-207	1561	IMF-238	1292	ORF-77	377	ORF-53	506
	5	MMF-214	1732	IMF-211	1459	MMF-213	1191	IMF-316	376	ORF-72	358
	6	MMF-234	1731	IMF-213	1412	ORF-154	831	ORF-126	371	ORF-178	350
	7	MMF-254	1405	MMF-208	1402	IMF-243	822	ORF-3	344	ORF-3	348
	8	ORF-203	1286	IMF-212	1354	ORF-77	758	ORF-120	307	MMF-209	340
	9	MMF-284	1215	ORF-50	987	ORF-166	758	MMF-303	305	IMF-319	325
	10	MMF-299	1086	ORF-164	986	ORF-143	640	ORF-127	285	MMF-212	322

4.5.5 RQ5: Impact of Main Parameters in MLPrior

Objectives: We delve into the impact of main parameters on the effectiveness of MLPrior.
Experimental design: Building upon the existing research by Wang *et al.* [2], We delve into an exploration of the impact of three main parameters within the MLPrior's ranking model. These parameters include *max_depth*, which denotes the maximum tree depth for each XGBoost model, *colsample_bytree*, representing the sampling ratio of feature columns during the tree construction process, and *learning_rate*, indicating the boosting learning rate utilized in the XGBoost ranking model. To achieve the research objectives, we conducted a series of experiments using natural datasets. We carefully modified the aforementioned three main parameters and observed the variations in the effectiveness of MLPrior (measured by APFD).
Results: The experimental results of RQ5 are presented in Figure 4.4, illustrating the fluctuations in MLPrior's effectiveness when the main parameters' values are altered. The X-axis represents the parameter values, while the Y-axis represents MLPrior's effectiveness (measured by APFD). The solid red line corresponds to MLPrior, while the dashed lines represent the confidence-based test prioritization approaches. We investigated the influence of the main parameter on MLPrior's effectiveness across both binary classification datasets (Adult, Bank, and Stroke) and multiclass classification datasets (Diabetes and Heartbeat).

MLPrior consistently outperforms the confidence-based test prioritization approaches, even when the values of the main parameters are altered. Notably, we see that MLPrior consistently outperforms the confidence-based test prioritization methods across all subjects, as evidenced by the red line persistently positioned above the blue dashed lines. For example, in Figure 4.4e), we observe that when the parameter *colsample_bytree* varies, MLPrior's APFD ranges from 0.86 to 0.88, whereas the confidence-based methods' APFD effectiveness is approximately 0.75. Moreover, under the multiclass dataset Heartbeat, when the parameter colsample_bytree changes, MLPrior's APFD ranges from around 0.84 to 0.85, whereas the APFD effectiveness of confidence-based methods ranges from around 0.745 to 0.750. Under the multiclass dataset Diabetes, when the parameter learning_rate changes, MLPrior's APFD ranges from around 0.77 to 0.78, whereas the APFD effectiveness of confidence-based methods is around 0.71.

The parameter colsample_bytree has a relatively small impact on the effectiveness of MLPrior, while the parameters max_depth and learning_rate have relatively high effects. Furthermore, we observe that the parameter *colsample_bytree*, which determines the sampling ratio of feature columns during the construction of each tree, has a relatively modest impact on the effectiveness of MLPrior. In other words, the effectiveness of MLPrior remains relatively stable even when the parameter *colsample_bytree* is altered. In contrast, the parameters *max_depth* (the maximum tree depth) and *learning_rate* (the boosting learning rate) exert a relatively high impact on the performance of MLPrior.

Answer to RQ5: *MLPrior consistently outperforms the confidence-based test prioritization approaches, even when the values of the main parameters are altered. The parameter colsample_bytree has a relatively small impact on the effectiveness of MLPrior, while the parameters max_depth and learning_rate have relatively high effects.*

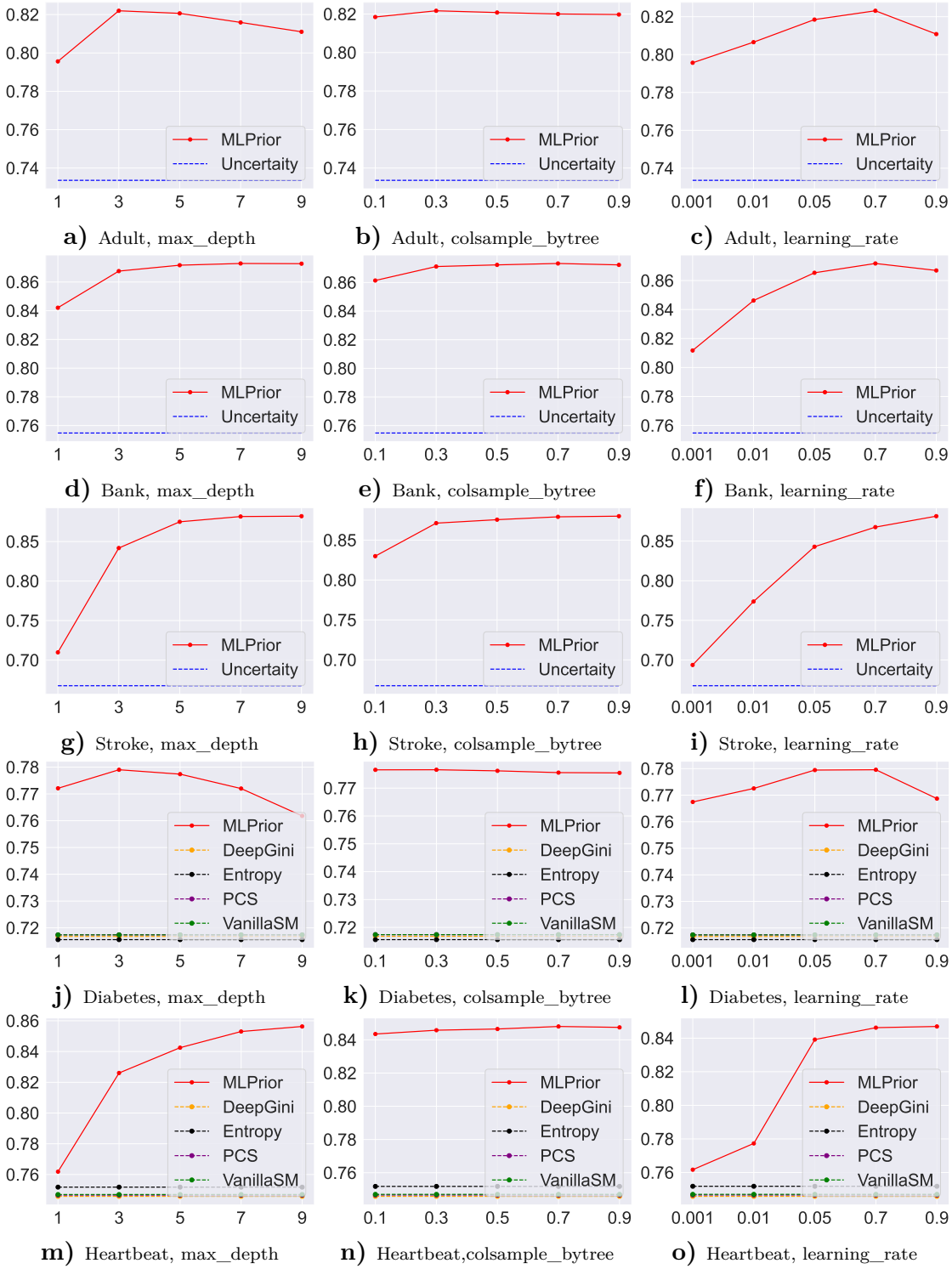


Figure 4.4: Impact of main parameters in MLPrior

4.6 Discussion

4.6.1 Generality of MLPrior

While we employed five ML models in our study, MLPrior can actually be adapted for a broad range of classical ML models through simple modifications to the model mutation rules, specifically by enabling them to target the architecture parameters or weight parameters of the evaluated model. We explain below why MLPrior exhibits generality. First, the core element of MLPrior is feature generation, which involves generating three essential types of features from the target tests: Model mutation features, Input mutation features, and Attribute features. Once the features are generated, MLPrior can utilize the ranking model to learn from these features for the purpose of test prioritization. Concerning model mutation features, making the aforementioned simple adjustments (i.e., enabling model mutation rules to target the architecture parameters or weight parameters of the evaluated model) can allow for the generation of model mutation features. For input mutation features and attribute features, MLPrior is capable of directly generating these features. Consequently, MLPrior can be applied to a diverse range of classical ML models.

Moreover, to better demonstrate the generality of MLPrior, we provide a detailed explanation of how to apply MLPrior to a new type of ML model.

- **Skills needed to apply MLPrior to new ML models** When an ML testing practitioner aims to apply MLPrior to a new type of ML model, they need to possess the following skills: 1) An understanding of the internal parameters and mechanisms of the new machine learning model, to effectively carry out model mutation operations in accordance with MLPrior's methodology; 2) Basic Python knowledge to replace the functions for model mutation of the new type model with those for the original model; 3) Since input mutation and attribute feature generation are already designed as automatic pipelines, the testing practitioner can directly execute them without needing additional skills.
- **Characteristics for models to utilize MLPrior** When a model exhibits the following characteristics, it can be added to the set of models that can use MLPrior: 1) The dataset of the model is in the tabular format, as our input mutation and attribute feature generation operations are specifically crafted for classical ML models that utilize tabular datasets; 2) The model is a white-box model, which allows for modifications to its internal structure or parameters, facilitating the implementation of MLPrior's model mutation operations.

Furthermore, we offer the following protocol to guide an ML testing practitioner in adapting MLPrior to new model classes. It details the systematic process for generating the model mutation features, original attribute features, and input mutation features.

- **Model Mutation Feature (MMF) Generation** To generate the model mutation feature for a test input, the following process should be executed: **1) Parameter Selection** Following the methodology of MLPrior's model mutation rules (i.e., modifying the architectural parameters or weight parameters of the model), the ML testing practitioner needs to select appropriate model parameters for the purpose of mutation and replace the previous model mutation function with the new model mutation function. **2) Automatic Pipeline** Once the replacement is complete, all other parts are automated pipelines, and MLPrior can automatically generate model mutation features.

- **Original Attribute Feature (OAF) Generation** The ML testing practitioner can directly obtain the OAF by implementing MLPrior, as we have designed an automated pipeline for the original feature transformation. This pipeline is capable of supporting new datasets in tabular format.
- **Input Mutation Feature (IMF) Generation** The ML testing practitioner can directly acquire the IMF by implementing MLPrior. MLPrior can automatically perform input mutations to obtain mutation features. This pipeline is capable of supporting new datasets in tabular format.

4.6.2 Threats to Validity

Threats to Internal Validity. The primary internal threats to the validity primarily pertain to the implementation of the compared approaches. To mitigate the threat, we implemented the compared approaches based on the implementations published by their respective authors. Another internal threat arises from the inherent randomness inherent in the training process of the ML models. To mitigate this potential issue, we conducted a statistical analysis. Specifically, we repeated all the experiments five times and reported the average experimental results. Furthermore, we calculated the p-value of the experimental results to demonstrate the stability of our findings.

Threats to External Validity. The external threats to validity arise from the ML models and test datasets employed in our study. To mitigate these threats, we carefully selected a variety of ML models and datasets that are utilized by several top-level conferences [66, 121, 135] in the field of ML testing. Moreover, our evaluation of MLPrior extended beyond natural datasets to encompass a spectrum of scenarios, encompassing mixed noisy datasets (comprising both natural and noisy data) as well as fairness-oriented datasets. This approach allowed us to substantiate the efficacy of MLPrior across various contexts.

4.7 Related Work

4.7.1 Test Prioritization Techniques

Test prioritization aims to establish an optimized sequencing of tests with the objective of early detection of system bugs. In the field of traditional software testing, numerous test prioritization approaches have been proposed [136, 137, 138, 139, 140]. Lou *et al.* [141] introduced an innovative approach to prioritize test cases, focusing on the inherent ability of individual test cases to detect faults. Their approach consists of two distinct models: a statistics-based model and a probability-based model, both of which quantify the fault detection capability of each test case. Through empirical evaluations, they demonstrated that the statistics-based model outperformed alternative methods, underscoring the significance of incorporating fault detection capability within the realm of test case prioritization. Henard *et al.* [142] conducted a thorough comparative study to analyze existing test prioritization techniques, finding that the difference between white-box strategies [143] and black-box strategies [144] are small. Chen *et al.* [145], in pursuit of enhancing the velocity of compiler testing, introduced the LET (Learning and Scheduling-based Test prioritization) framework. This pioneering framework is underpinned by two salient processes: the learning process, designed to discern program features and prognosticate the potential of a novel test program in revealing bugs, and

the scheduling process, which strategically prioritizes test programs based on their propensity to unveil bugs.

In addition to the traditional field of software engineering, multiple test input prioritization strategies have been proposed in the literature for Deep Neural Networks (DNNs) [146, 2, 3, 7] to tackle the labeling-cost issue. Feng *et al.* [3] introduced DeepGini, which prioritizes tests by utilizing the Gini score to measure model confidence for each test input. Byun *et al.* [147] assessed various white-box metrics for ranking bug-revealing inputs, encompassing widely-used measures like softmax confidence, Bayesian uncertainty, and input surprise. Furthermore, Weiss *et al.* [7] extensively investigated diverse test input prioritization techniques for DNNs, particularly focusing on uncertainty-based metrics such as Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy. These metrics have demonstrated effectiveness in identifying potentially misclassified test inputs and have played a crucial role in facilitating test prioritization endeavors. Furthermore, Wang *et al.* [2] proposed a mutation-based test prioritization approach for DNNs, which will be described in the subsequent Section 4.7.4.

4.7.2 DNN Testing

In addition to test prioritization, the domain of DNN testing encompasses several other pivotal areas, such as test selection [35, 36], test input generation [4, 148], and test adequacy. Test selection aims to select a representative subset from the original test set to estimate the accuracy of the entire test set. Various test selection approaches have been proposed in the literature. Li *et al.* [36] proposed CES (Cross Entropy-based Sampling), which performs test selection by minimizing the cross-entropy between the selected test set and the entire test set, ensuring that the distribution of the selected test set closely matches the original set. Chen *et al.* [35] proposed PACE, which selected representative test inputs based on clustering, prototype selection, and adaptive random testing. First, Pace divides all test inputs into clusters based on their testing capabilities. Then, PACE utilizes the MMD-critic algorithm [37] to select prototypes from each group. For tests not belonging to any groups, PACE leverages adaptive random testing [149] to select test inputs by considering diversity.

Within the domain of test input generation, researchers have proposed a multitude of techniques aimed at generating diverse and effective inputs for DNN systems. Pei *et al.* [4] proposed DeepXplore, a white-box differential technique that focuses on generating test inputs capable of effectively evaluating the robustness of real-world DL systems. By leveraging the notion of neuron coverage, DeepXplore generates inputs that cover distinct regions of the neural network. Tian *et al.* [148] presented DeepTest, a method specifically tailored for generating test inputs to assess the performance of autonomous driving systems. DeepTest employs a greedy search strategy in conjunction with nine realistic image transformations to produce a diverse set of challenging input data. By systematically exploring the input space, DeepTest aims to uncover potential failures or limitations in autonomous driving systems, thereby enhancing their safety and reliability.

Regarding test adequacy, Ma *et al.* [5] proposed a set of multi-granularity testing criteria, including k-multisection neuron coverage, neuron boundary coverage, and strong neuron activation coverage. These criteria have been developed to identify corner behaviors and uncover potential vulnerabilities in DNN systems by compre-

hensively examining the coverage of various aspects of the neural network's behavior. Kim *et al.* [49] introduced surprise adequacy as a novel test adequacy criterion for testing DL systems. The surprise adequacy criterion emphasizes the importance of a test input being both challenging and informative while still adhering reasonably to the underlying training data distribution. This criterion emphasizes that a good test input should be sufficiently challenging and informative but should not deviate excessively from the training data distribution.

4.7.3 Mutation-based Test Prioritization for Traditional Software

Mutation testing [38] entails generating intentional defects, referred to as mutants, within the software code to assess the test suite's quality. In the field of traditional software testing [141, 150, 64], mutation testing can be employed to assess the fault-detection capabilities of individual test cases, thereby achieving test prioritization. Lou *et al.* [141] introduced a novel test-case prioritization approach that determines the order of test cases by considering their fault detection ability. This ability is defined based on the analysis of mutation faults simulated from real software faults. By strategically ordering the test cases, this approach aims to maximize the efficiency of the testing process by prioritizing the detection of critical faults. Papadakis *et al.* [64] conducted a mutation analysis as an alternative technique to Combinatorial Interaction Testing (CIT). Their research suggests that the mutants generated using their approach demonstrate a stronger correlation with code-level faults than the input interactions targeted by the CIT approach. This underscores the potential of mutation analysis to offer valuable insights into underlying faults within software systems and guide test case prioritization. Furthermore, Shin *et al.* [150] proposed a novel test case prioritization method that combines mutation-based and diversity-based approaches. They demonstrate that mutation-based prioritization is as effective as, or more effective than, random prioritization and coverage-based prioritization.

4.7.4 Mutation Testing and Mutation-based Test prioritisation for Deep Learning

Mutation Testing for DNNs The field of mutation testing for DNNs has seen significant exploration, with numerous studies contributing to the evolution of various mutation operators and frameworks [48, 151, 146]. A notable contribution in this domain is from Ma *et al.* [48], who introduced DeepMutation. This innovative approach is designed to assess the quality of test data for DL systems through comprehensive mutation testing. DeepMutation encompasses a diverse array of mutation operators at both the source and model levels. These operators are meticulously crafted to inject faults into different components of DL systems, including training data, programming code, and the models themselves. Building upon this foundation, Hu *et al.* further expanded their work with the development of DeepMutation++ [151], an advanced mutation testing tool specifically tailored for DL systems. DeepMutation++ introduced a set of new mutation operators that are particularly suited for feed-forward neural networks (FNNs) and Recurrent Neural Networks (RNNs). A key feature of this tool is its capability to dynamically mutate the runtime states of RNNs, a critical aspect for evaluating the resilience of these networks under various operational conditions. Humbatova *et al.* [146] made a significant stride in the field by developing DeepCrime, the first mutation testing tool that implements

DL mutation operators grounded in actual DL faults. DeepCrime is characterized by its comprehensive set of 24 newly defined mutation operators. These operators are not just theoretical constructs but are based on real-world faults observed in DL systems, making DeepCrime a highly practical tool for testing and improving the reliability of these systems.

Mutation-based test prioritization for DNNs Wang *et al.* [2] introduced PRIMA, an innovative test input prioritization technique founded on intelligent mutation analysis. PRIMA is applicable to both classification and regression models and possesses the capability to handle test inputs generated through adversarial input generation techniques, thereby enhancing the probability of misclassification. However, PRIMA's model mutation rules cannot be adapted to classical ML models.

In this study, we proposed MLPrior, a mutation-based test input prioritization approach specifically designed for classical ML models. The significant differences between MLPrior and PRIMA are as follows:

- **Different Approaches for Model Mutation** MLPrior and PRIMA leverage different model mutation approaches. In **MLPrior**, model mutations are specifically designed for white-box classical machine learning models. These mutations are based on the interpretable nature of these models and involve modifying the architecture parameters or weight parameters of the evaluated model. **PRIMA**, on the other hand, is primarily focused on DNNs, which are non-interpretable black-box models. Examples of model mutations in PRIMA include adding noise to the weights of neurons and altering the structure of DNN layers.
- **Attribute Feature Inclusion** Another significant difference is that **MLPrior** employs the inherent attribute features of classical ML model datasets for test prioritization. In contrast, **PRIMA** does not incorporate this information into its test prioritization procedure. The motivation behind MLPrior's utilization of attribute features for test prioritization is that classical ML datasets typically exhibit lower-dimensional features compared to DNN test data. Additionally, these features are carefully selected by domain experts, directly reflecting the attribute information associated with each test input.
- **Feature Generation Strategy** In terms of model and input mutation, compared to PRIMA, **MLPrior** emphasizes generating mutation features directly from mutation results. For example, in model mutation, the i_{th} element in the vector indicates whether the i_{th} mutated model is 'killed' by this input. This method is intuitive and reproducible.
- **Use of Multiple Ranking Models** MLPrior employs five different ranking models and assesses their effectiveness in utilizing mutation features for test prioritization. In contrast, PRIMA utilizes only a single ranking model. By comparing multiple ranking models, MLPrior can identify the most effective model for learning mutation features in the context of test prioritization.

4.8 Conclusion

In order to solve the labeling cost problem for classical ML models, we propose MLPrior, which prioritizes tests that are more likely to be misclassified. MLPrior leverages the unique characteristics of classical ML classifiers, including their interpretability and carefully engineered dataset features, to effectively prioritize test inputs. The foundational principles of MLPrior are twofold: Firstly, tests exhibiting higher sensitivity to mutations are more likely to be misclassified. Secondly, tests

situated closer to the decision boundary of the model are more susceptible to misclassification. Capitalizing on these principles, we design mutation rules specifically for classical ML models and their datasets. For each test, we generate mutation features while simultaneously transforming its attribute into a feature vector that can indirectly quantify the proximity between it and the decision boundary. Concatenating these features, MLPrior constructs a final vector for each test, which will be inputted into a pre-trained ranking model for the purpose of predicting its misclassification probability. Finally, MLPrior ranks all the tests according to their misclassification scores in descending order. We conducted an extensive study to evaluate MLPrior, utilizing 185 different types of subjects that encompass natural, noisy, and fairness datasets. The experimental results demonstrate that MLPrior exhibits higher effectiveness compared to existing test prioritization methods, yielding an average improvement of 14.74%∼66.93% on natural datasets, 18.55%∼67.73% on mixed noisy datasets, and 15.34%∼62.72% on fairness datasets.

Availability

Our replication package is available at

`https://github.com/yinghuali/MLPrior`.

5 GraphPrior: Mutation-based Test Input Prioritization for Graph Neural Networks

In this chapter, we propose a novel test prioritization approach called GraphPrior, which is specifically designed to prioritize test inputs that are more likely to be misclassified by Graph Neural Networks (GNNs). GraphPrior addresses a fundamental challenge in GNN testing: existing test prioritization methods developed for Deep Neural Networks (DNNs) fail to consider the dependencies among test inputs (nodes) in graph-structured datasets. Inspired by mutation testing, GraphPrior generates mutated GNN models and prioritizes test inputs that "kill" many mutations, as they are more likely to be misclassified. It employs two ranking methods: a killing-based approach, treating all mutations equally, and a feature-based approach, which learns the importance of different mutations via ranking models. By effectively prioritizing potentially misclassified test inputs, GraphPrior enhances fault detection and debugging efficiency for GNNs.

This chapter is based on the work published in the following research paper:

- Xueqi Dang, Yinghua Li, Mike Papadakis, Jacques Klein, Tegawendé F. Bissyandé, Yves Le Traon. GraphPrior: Mutation-based Test Input Prioritization for Graph Neural Networks. ACM Transactions on Software Engineering and Methodology (TOSEM). Accepted for publication on Jun. 13, 2023

Contents

5.1 Introduction

In recent years, graph machine learning [152, 153] has been widely adopted for modeling graph-structured data. In this realm, the emergence of graph neural networks (GNNs) [154] has offered promising results in diverse domains, such as recommendation systems [155, 156, 157], social network analysis [158, 159, 160], and drug discovery [161, 162]. GNNs, like typical neural networks [163] [164], are abstractions of the underlying data. Thus, their inference can suffer from faults [165] [166] [167], which can lead to severe prediction failures, especially in security-critical use cases. Testing is considered to be a fundamental practice that is widely adopted to ensure the performance of neural networks, including GNNs. However, like traditional deep neural networks (DNNs), GNN testing also suffers from the lack of automated testing oracles, which necessitates the manual labeling of test inputs. However, this labeling process can require significant human effort, especially for large and complex graphs. Moreover, in certain specialized domains, such as the protein interface prediction [168] of drug discovery, labeling intensively relies on domain-specific knowledge, further increasing its costs.

Prior works [147, 3, 2, 49] have focused on *test prioritization* to relieve the labeling-cost problem for DNNs. Test prioritization approaches aim to prioritize test inputs that are more likely to be misclassified (i.e., fault-revealing test inputs) so that such inputs can be identified earlier to reveal system bugs. Existing approaches are mainly divided into two categories: coverage-based and confidence-based test prioritization approaches. Coverage-based approaches prioritize test inputs based on neuron coverage through adapting coverage-based prioritization methods from traditional software testing [92, 169]. Confidence-based approaches assume that test inputs for which the model is less confident are more likely to be misclassified and thus should be prioritized higher. Feng *et al.* [3] proposed the state-of-the-art confidence-based approach DeepGini, which considers that a test input is more likely to be misclassified by a DNN model if the model outputs similar prediction probabilities for each class. More recently, Wang *et al.* [2] proposed PRIMA, which leveraged mutation analysis and learning-to-rank methods to prioritize test inputs for DNNs. However, despite its effectiveness in DNN test prioritization, PRIMA cannot be directly applied to GNNs since their mutation operators are not adapted to graph-structured data and GNN models.

Furthermore, existing studies [47] have focused on metrics for data selection (e.g., margin and least confidence), which can also be used to detect possibly-misclassified test data. Although the aforementioned approaches have been demonstrated to be effective for DNN models in some cases, they have the following limitations when applied to GNN models:

- First, to the best of our knowledge, current coverage-based approaches do not provide interfaces for GNN models and thus cannot be directly applied. Moreover, existing research [3] has demonstrated that coverage-based approaches are not effective compared to confidence-based approaches.
- Second, despite the effectiveness of confidence-based approaches on traditional DNNs, they do not take into account the interdependencies between test inputs of GNNs, which are particularly crucial for GNN inference. In other words, GNN test inputs are typically represented as graph-structured data consisting of nodes and edges, while confidence-based prioritization approaches usually deal with test

sets in which each test is independent and has no connections with others.

- Third, the effectiveness of uncertainty-based metrics can be limited when facing some specific adversarial attacks. If the aim of an attack is to generate test inputs that maximize the probability of incorrect classification, then the utility of uncertainty metrics can be limited. This is because the underlying assumption of uncertainty-based metrics is that: if a model is more uncertain about classifying a test, this test is more likely to be misclassified. However, in such scenarios, even if a model is confident on a test, this test can still have a high probability of being misclassified.

To overcome the aforementioned problems, in this paper, we propose GraphPrior (**GNN**-oriented Test **Prior**itization), a set of test prioritization approaches specifically for GNNs. GraphPrior identifies and prioritizes possibly-misclassified test inputs via mutation analysis. Given a test set for a GNN model, GraphPrior regards a test input that kills more mutated models (i.e., variants of the original GNN model that is slightly changed) of the original GNN model as more likely to be misclassified. Here, killing means the prediction result to the test input via the GNN model and the mutated model is different. To this end, we design a set of mutation rules to generate mutated models specifically for GNNs by slightly changing the training parameters of the original model. After obtaining the mutation results of each test input, GraphPrior introduces several ranking models (ML/DL models) [14, 112, 69] to rank the test set. The working principle of GraphPrior is inspired by mutation testing research as this has been realized for both model-based [62, 63, 64] and code-based [96, 170, 95] testing. The key underlying principle in all cases is that test cases that distinguish the behavior of mutants from that of the original artifact are useful and more likely to detect other underlying faults [171, 64, 62].

While both the GraphPrior and PRIMA (i.e., the state-of-the-art DNN test prioritization approach) use mutation analysis, GraphPrior differs from PRIMA in terms of its mutation rules, feature generation, and ranking models: 1) GraphPrior's mutation rules can directly or indirectly affect the message passing between nodes in graph data. In contrast, the mutation rules of PRIMA are designed for traditional DNNs, where the test inputs are independent, and therefore, the mutation rules do not affect the relationships between tests; 2) GraphPrior generates a mutation feature vector for each test input based on its mutation results, where the i_{th} element in the vector denotes whether the i_{th} mutated model is killed by this input. This feature generation strategy is intuitive and reproducible. In addition to this, the generation method exhibits several other advantages. First, by using binary indicators (1 or 0) as elements of the mutation feature vector, the information is transformed into a concise vector representation. Second, the fine-grained nature of the mutation feature vector allows for a detailed analysis of the effects of individual mutations. In particular, further analysis can be conducted to assess the contributions of each mutated model to GraphPrior. By tracing back to the corresponding mutation rules for the top critical mutated models, we can gain insights into which mutation rules made higher contributions to GraphPrior. The experimental results demonstrate its effectiveness; 3) GraphPrior employs five ranking models and compares their effectiveness in utilizing mutation features for test prioritization, while PRIMA only uses a single ranking model. By comparing multiple ranking models, GraphPrior can identify the optimal ranking model for learning mutation features in test prioritization.

GraphPrior has broad applicability across a wide range of contexts, including

software development, scientific research, and financial systems. For instance, GraphPrior can be employed to gain insights into the vulnerabilities of GNN models used in financial transaction fraud detection. In this specific context, where nodes represent accounts and edges represent transaction transfers, the first step is to utilize the GNN model under test to identify a group of potentially fraudulent accounts. Subsequently, these identified accounts serve as test inputs for GraphPrior. By prioritizing accounts that are more likely to be misclassified by the model (i.e., accounts falsely classified as fraudulent), GraphPrior places them at the top of the recommendation list. Consequently, by labelling and analyzing these bug-revealing tests earlier, the fraud analysis team can unveil the bugs and vulnerabilities of the GNN model more efficiently.

It is important to note that, GraphPrior is specifically designed for GNNs, and its impact on DNNs has not been evaluated. This is because in graph datasets, nodes are interconnected, and the mutation rules of GraphPrior can directly or indirectly affect the message passing between nodes in the prediction process. In contrast, in traditional DNNs, each sample in a dataset is typically independent, and as a result, such mutation rules are unlikely to affect the transmission of information between tests. Therefore, the effectiveness of GraphPrior's mutation rules for DNNs remains uncertain, as no related experiments have been conducted to evaluate it.

We conducted an extensive study to evaluate the performance of GraphPrior based on 604 subjects. Here, a subject refers to a pair of graph dataset and GNN model. We compare GraphPrior with 6 uncertainty-based metrics [7] [3] [172] that can be used to prioritize possibly-misclassified test inputs and adopt random selection as the baseline method. Our experimental results demonstrate that GraphPrior performs well across all subjects and outperforms the compared approaches on average.

As mentioned before, one essential problem of confidence-based approaches is that adversarial attacks may lead to a model being more confident in the incorrect prediction, resulting in the failure of the approach. Therefore, we also evaluate GraphPrior on test inputs generated from graph adversarial attacks of existing studies [173, 174, 175, 176]. Furthermore, since the effectiveness of test prioritization methods may vary depending on the degree of the adversarial attack, we set different attack levels to generate adversarial data and compared GraphPrior with the compared approaches. In addition to the evaluation of GraphPrior, we compare the effectiveness of different mutation rules in generating top contributing mutated models, aiming to identify which mutated rules contribute more to each GNN model. In the last step, we investigate whether GraphPrior and the uncertainty-based metrics can select informative retraining tests to improve a GNN model. Our experimental results demonstrate that GraphPrior achieved better effectiveness compared with the uncertainty-based test prioritization methods. We publish our dataset, results, and tools to the community on Github[1].

Our work has the following major contributions:

- **Approach.** We propose GraphPrior, a set of mutation-based test prioritization approaches for GNNs. To this end, we design a set of mutation rules that mutate GNN models by slightly changing their training parameters. We carefully select ranking models to analyze the mutation results for effective test prioritization.
- **Study** We conduct an extensive study based on 604 GNN subjects involving

[1]https://github.com/yinghuali/GraphPrior

natural and adversarial test sets. We compare GraphPrior with existing DNN approaches that could detect possibly misclassified test inputs. Our experimental results demonstrate the effectiveness of GraphPrior.

- **Mutation rule analysis** We compare the effectiveness of the GNN mutation rules in generating top contributing mutated models, observing that the mutation rule HC (i.e., mutating Hidden Channels) makes top contributions to most GNN models in test input prioritization.

5.2 Background

In this section, we introduce the key domain concepts for our work, including Graph Neural Networks and Test Input Prioritization for DNNs.

5.2.1 Graph Neural Networks

Graph neural networks (GNNs) have achieved great success in handling machine learning problems on graph-structured data [177] [157] [178]. Unlike traditional neural networks running on fixed-sized vectors, GNNs deal with graphs of varying sizes and structures. Therefore, GNNs can capture complex relationships between data points and make more accurate predictions. GNNs have been used in a wide range of tasks, including recommendation system [157, 156, 179], protein-protein interaction (PPI) prediction [180, 181, 168] and traffic forecasting [182, 183, 184].

Graphs A graph is a data structure consisting of two components: nodes (vertices) and edges. A graph H can be defined as $H = (V, E)$, where V is the set of nodes, and E are the edges between them. In a graph, nodes can represent entities (e.g., persons, places, or things), while the edges define the relationships between nodes. The edges can be either directed or undirected based on the directional dependencies that exist between nodes. Graphs can be utilized to model complex systems such as social media networks, molecular structures, and citation networks. For example, in the context of citation networks, publications can be represented as nodes, and the citations between them can be represented as edges. Graph datasets are collections of graph data that can be used to train and evaluate GNNs. Some benchmark graph datasets [185] include Cora, CiteSeer, and PubMed. In this paper, we evaluated GraphPrior and the compared approaches on several graph datasets obtained from existing studies [186] [187].

Graph Embeddings Graph embedding [188] is an approach used to transform nodes, edges, and their associated features into lower dimensional representation while maximally preserving the graph structural information and graph properties. Graph analytics methods usually suffer from high computational and storage costs, limiting their applicability in real-world scenarios. The use of graph embedding has shown promising results as an efficient and effective way to address the graph analytics problem.

Message Passing Scheme In GNNs, the message-passing scheme is commonly employed [189], whereby nodes aggregate and transform the information from their neighbors in each layer. Through stacking multiple GNN layers, this mechanism facilitates the propagation of information across the entire graph structure, allowing for the effective embedding of nodes into low-dimensional representations. These node representations may subsequently be leveraged by a differentiable prediction layer, thereby enabling end-to-end training of the complete model.

GNN models A graph neural network (GNN) model is a type of neural network

designed to operate on graph data structures. Typically, a GNN model contains two crucial parts: a graph convolution layer [163] to capture the relationship between nodes in the graph and a classifier [190] to make predictions based on the captured relationship. In general, a GNN model takes graph-structured data as inputs and produces outputs based on its corresponding task. For example, the output for a GNN model that deals with node-level tasks (i.e., GNN tasks that are concerned with predicting the identity or role of each node within a graph) is typically a prediction for nodes in the input graph. In this paper, we evaluated our proposed test prioritization approach, GraphPrior, and the compared approaches on various GNN models [163, 185, 191, 192] that deal with node classification tasks.

Graph Adversarial Attacks Graph adversarial attacks [175] [193] [194] [195] involve the manipulation of graph structure or node features to generate graph adversarial perturbations that can fool GNN models. This vulnerability of GNNs has raised serious concerns regarding their reliability and safety, particularly in safety-critical applications such as financial systems and risk management. For instance, in a credit scoring system, attackers can exploit the vulnerability of GNNs to create fake connections with high-credit customers to evade fraud detection models. In this paper, we applied eight graph adversarial attacks from existing studies [173, 174, 175, 176] to generate adversarial inputs for the evaluation of GraphPrior.

5.2.2 Test Input Prioritization for DNNs

In DNN testing, test input prioritization aims to prioritize tests that are more likely to be misclassified (i.e., bug-revealing test inputs) by the DNN model. In this way, more important test inputs can be labeled earlier in a limited time, which can improve the efficiency of DNN testing. In the literature, several prioritization approaches have been proposed to deal with the labeling-cost issues [3, 2, 147, 196].

The majority of approaches for prioritizing tests in Deep Neural Networks (DNNs) can be classified into two categories, coverage-based and confidence-based [2]. Confidence-based approaches, such as DeepGini [3], prioritize test inputs based on the model's confidence. Specifically, these methods identify inputs that are likely to be incorrectly predicted by the DNN model, given that the model outputs similar probabilities for each class. In contrast, coverage-based approaches, such as CTM [92], simply extend traditional software system testing methods to DNN testing, and have been shown to underperform compared to confidence-based approaches [3]. Weiss *et al.* [7] conducted a comprehensive investigation of the capabilities of various DNN test input prioritization techniques, including some notable uncertainty-based metrics such as Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy. The Vanilla Softmax metric is calculated as the highest activation in the output softmax layer for a classification problem, subtracted from 1. PCS, on the other hand, is defined as the difference in softmax likelihood between the predicted class and the second runner-up class. Additionally, Entropy is considered as an alternative metric in the softmax layer proposed by the authors of DeepGini. These metrics have been demonstrated to be effective in identifying possibly-misclassified test inputs, and can aid in guiding test prioritization efforts.

The aforementioned uncertainty-based test prioritization can be adapted for test input prioritization for GNNs. GraphPrior differs from these approaches in that GraphPrior leverages mutation analysis to perform test prioritization. The mutation analysis of GraphPrior exploits the specific properties of GNNs. Specifically,

GraphPrior's mutation rules can directly or indirectly affect the message passing between nodes in a graph. In contrast, uncertainty-based approaches rely on the prediction uncertainty of the DNN model to prioritize test inputs without accounting for the interdependence between nodes.

Currently, the state-of-the-art technique for DNN test prioritization is PRIMA, which prioritizes fault-revealing test inputs based on mutation analysis. However, PRIMA is not suitable for GNN test prioritization because: 1) its input mutation rules are specifically designed for DNN testing datasets where each sample is independent of each other. In contrast, graph datasets have complex interdependence between nodes, making PRIMA unsuitable for test prioritization in this context; 2) GNNs employ graph operations and message passing mechanisms to aggregate and update information from neighboring nodes, thereby facilitating improved representation and learning within graph structures. The model mutation rules employed in PRIMA are not suitable for accommodating the graph operation mechanisms intrinsic to GNNs.

In addition to the aforementioned test prioritization techniques, several active learning [172] methods can also be adapted to prioritize DNN tests, such as Least Confidence and Margin. Active learning aim to selects the most informative samples to be labeled by a human expert. When applied to test prioritization, active learning can be used to identify the most critical and informative test cases that can reveal bugs in the system.

5.3 Approach

5.3.1 Overview

In this paper, we propose GraphPrior, a set of test prioritization approaches for GNNs to prioritize test inputs. GraphPrior consists of six mutation-based test prioritization approaches: KMGP, LRGP, RFGP, LGGP, DNGP and XGGP. These approaches are discussed later in this Section. We present the overview of GraphPrior in Figure 5.1, in which the input of GraphPrior is a GNN test set, and the output is the test set that has been prioritized. Given a test set T for a GNN model G, the implementation process of GraphPrior is presented as follows.

Generating mutants for the GNN model First, GraphPrior generates mutated models (i.e., mutants) for the GNN model G based on carefully designed mutation rules (cf. Section 5.3.2).

Obtaining mutation results through killing mutants For each test input, GraphPrior identifies which mutated models it kills. Here, a mutated model is killed by a test input if the prediction results of this input via the mutated model and the original model G are different. In this way, GraphPrior obtains the mutation result of each test input.

Generating feature vectors from the mutation results For each test input, GraphPrior generates a mutation feature vector for it based on its mutation results. The i_{th} element of this feature vector denotes whether this input kills the i_{th} mutated model. More specifically, given a test input $t \in T$, if t kills a mutated model M_i, then the i_{th} element of t's mutation feature vector is set to 1. Otherwise, the i_{th} element is set to 0.

Ranking test input based on mutation feature vectors via ranking models GraphPrior utilizes ranking models [14, 112, 69] to calculate a misclassification score

for each test input based on its feature vector. This score can indicate how likely a test input will be misclassified by the GNN model. Finally, GraphPrior ranks them based on their misclassification scores in descending order and outputs the prioritized test set T'.

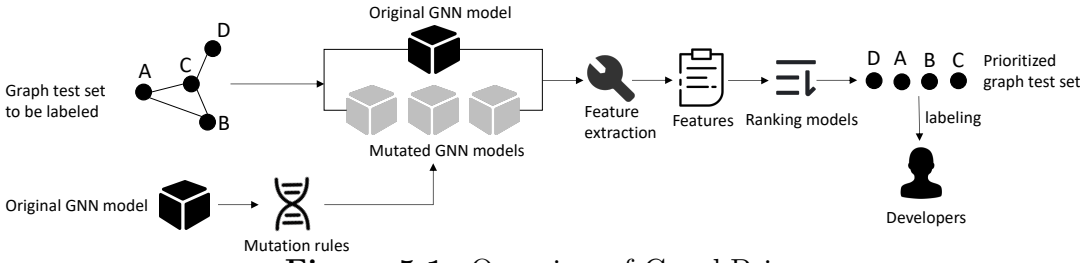


Figure 5.1: Overview of GraphPrior

5.3.2 Mutation Rules

In GraphPrior, mutation rules are employed to generate mutated models of a GNN model by making slight changes to its training parameters. We select the following parameters because they can impact the message passing in the GNN prediction process. More specifically, in the mutated GNN model, the manner in which nodes acquire information from their neighboring nodes is slightly different from that of the original GNN model. Although variations of GNNs can be obtained even without changing training parameters, the resulting model mutants cannot produce meaningful differences in the GNN model's behavior. By changing the selected training parameters to generate mutants, we can intentionally introduce meaningful modifications to the model's behavior in terms of the interdependencies between nodes during the prediction process. We present all the mutation rules of GraphPrior as follows.

- **Self Loops (SL)** [163, 185] SL is a Boolean parameter, which controls whether to add self-loops to the input graph. When the SL parameter is set to True, self-loops are introduced to each node in the graph. By incorporating self-loops, the inherent information of nodes can be effectively aggregated into their representation vectors, leading to a change in the weighting of their neighboring nodes, and thus affecting the interdependence of nodes in the prediction process.
- **Bias (BIA)** [163, 185, 191] BIA is a Boolean parameter, which determines whether to introduce a predetermined offset to the representation vectors of nodes. When the BIA parameter is enabled (set to True), each node will be assigned a corresponding bias parameter to its representation vector, allowing the GNN model to better capture the inherent properties of the graph and improve the interdependence between nodes in the prediction process.
- **Cached (CA)** [163] CA is a Boolean parameter that controls whether to cache the computation of node embeddings during the forward pass. When the CA parameter is set to True, the node embeddings are cached and reused during the backward pass to save computation time. Caching the computation of node embeddings can affect the interdependence between nodes by altering the order and efficiency of message passing.
- **Improved (IMP)** [163] IMP is a Boolean parameter that controls whether to use the improved message passing strategy, thus affecting the interdependence between nodes in the prediction process.
- **Normalize (NOR)** [191, 192] NOR is a Boolean parameter, which determines

whether to normalize the messages passed between nodes in the prediction process. When this parameter is set to "True," the messages are normalized by the number of neighbors that a node has before being passed to the next layer. This normalization can impact the contribution of each neighbor to the node's final representation, thus affecting the message passing between nodes in the prediction process.

- **Concat (CON)** [185] CON is a Boolean parameter, which controls how the representations of neighboring nodes are combined during message passing. When it is set to True, the representations of neighboring nodes are concatenated before being passed, resulting in a more expressive representation of the nodes, enabling the GNN to capture more nuanced interdependencies between them.
- **Heads (HDS)** [185] HDS is an integer parameter that determines the number of attention heads used in multi-head attention. Increasing the number of heads allows the model to capture more complex interdependence among nodes in the graph. Each attention head can focus on a different aspect of the node neighborhood, enabling the model to learn different representations of the graph.
- **Epoch (EP)** [185, 191, 192] EP is an integer parameter that controls the number of times a GNN model iterates over the training dataset. By increasing the number of epochs, a GNN model can better capture the interdependence between nodes for model inference.
- **Hidden Channel (HC)** [163, 185, 191, 192] HC is an integer parameter, which controls the dimensionality of the hidden representation in each layer of the GNN. Therefore, changing this parameter can impact the interdependence between nodes in a graph by enabling the GNN to learn more expressive node embeddings.
- **Negative Slope (NS)** [185] NP is a float parameter, which controls the slope of the negative part of the activation function used in the Gated Linear Unit (GLU) operation. GLU is a common non-linear function used in GNNs for message passing. Specifically, the GLU operation is used to combine the node features with the weighted sum of their neighboring nodes' features, which is the message passed between nodes in the graph. The negative slope parameter determines the slope of the activation function for negative input values in the GLU operation, thus impacting the message passing between nodes.

Based on the above mutation rules, for a given test set and a GNN model, GraphPrior generates N mutated models of the original model. We consider that a test input kills a mutated model if the predictions for this input via the mutated models and the original GNN model are different. Based on it, GraphPrior obtains the mutation results of all the test inputs.

Considering that the primary objective of generating mutated models is to obtain informative features for test prioritization, a statistical analysis is employed to validate their effectiveness. To achieve this, a series of repeated experiments are conducted, as outlined in Section 6.5. The results of these experiments demonstrate that GraphPrior's effectiveness is statistically significant, thereby confirming the statistical validity of the generated mutated models for the purpose of test prioritization.

5.3.3 Killing-based GraphPrior

This section presents the workflow of KMGP, the **K**illing **M**utants-based **G**NN Test **P**rioritization approach. Notably, KMGP operates on a "killing-based" principle, where test inputs that can kill more mutated models are considered as more likely to be misclassified and will be prioritized higher. It is worth noting that KMGP

assigns equal importance to each mutated model in the process of test prioritization, a distinct feature that distinguishes it from feature-based approaches, which will be elaborated upon in subsequent sections. Given a GNN model G, and a test input set $T = \{t_1, t_2, \ldots, t_n\}$, the detailed execution of KMGP can be divided into three key stages: mutation generation, killing-based mutation analysis, and test prioritization.

Mutation generation In the mutation generation stage, a group of mutated models $\{G'_1, G'_2, \ldots, G'_N\}$ are generated for the original GNN model G.

Killing-based mutation analysis This stage involves obtaining the mutation results of each test input $t \in T$ using the process outlined in Section 5.3.2. Subsequently, KMGP counts the number of mutants killed by each test input based on their mutation results.

Test prioritization In the third stage, KMGP prioritizes all the test inputs in T based on the number of mutated models they killed, with those that kill more mutants being prioritized higher in the test sequence.

5.3.4 Feature-based GraphPrior

In comparison to the killing-based GraphPrior approach, the feature-based approaches are characterized by automatic mutation feature analysis. This process involves the generation of mutated feature vectors based on the execution of mutated models, followed by the use of ranking models (ML/DL models), which assign different importance to each mutated model for test prioritization.

Overall, the feature-based approaches' workflow entails three key stages: mutated model generation, mutation feature generation, and learning-to-rank.

❶ **Mutated model generation** Given a GNN model G and a test set T, during the first stage, the feature-based approaches generate a group of mutated models (denoted as $\{G'_1, G'_2, \ldots, G'_N\}$) of the GNN model G based on the mutation rules specified in Section 5.3.2.

❷ **Mutation feature generation** Subsequently, the feature-based approaches associate a feature vector V_t of size N with each test input t, where N represents the number of mutated models, and $v_k (= V_t[k])$ maps to the execution output for the mutated model G'_k. If t kills the mutated model G'_k (i.e., the prediction results for t via the mutated models G'_k and the original model G are different), v_k is set to 1. Otherwise, it is set to 0.

❸ **Learning-to-rank** In the final stage, the feature-based approaches input the mutation features of each test input to the ranking model (ML/DL models) [112] [8] [69] [197] [14]. The ranking models can automatically learn different importance for each mutation feature to output misclassification scores. Here, each mutation feature corresponds to the execution result of a mutated model so that we can consider that the ranking models learn the importance of each mutated model for test prioritization. Finally, the feature-based approaches rank all the test inputs based on their misclassification scores in descending order.

In our study, we propose five feature-based GraphPrior approaches, which follow the similar workflow described above, but leverage different ranking models. These five approaches are XGGP (**XG**Boost-based **G**NN Test **P**rioritization), LRGP (**L**ogistic **R**egression-based **G**NN Test **P**rioritization), LGGP (**L**ight**G**BM-based **G**NN Test **P**rioritization), RFGP (**R**andom **F**orest-based **G**NN Test **P**rioritization) and DNGP (**DN**N-based **G**NN Test **P**rioritization). We briefly introduce the basic principle of the ranking models of these approaches as follows.

1) XGGP leverages the XGBoost algorithm [8] as the ranking model. XGBoost is a highly effective gradient boosting algorithm that combines decision trees to enhance the accuracy of predictions. XGGP utilizes the XGBoost algorithm to predict the misclassification score for a given test input based on its mutation features. This score reflects the likelihood that the input will be misclassified by a GNN model.
2) LRGP leverages the Logistic Regression algorithm [14] as the ranking model. Logistic regression leverages a logistic function to model the association between a categorical dependent variable and one or more independent variables.
3) LGGP leverages the LightGBM algorithm [69] as the ranking model. LightGBM is a gradient boosting framework that employs tree-based learning algorithms. The fundamental principle of LightGBM is similar to XGBoost, which employs decision trees based on learning algorithms. However, LightGBM introduces a novel optimization in the framework, with a primary focus on enhancing the speed of model training.
4) RFGP leverages the random forest algorithm [112] as the ranking model. Random Forest is an ensemble learning algorithm that constructs multiple decision trees using random subsets of the training data and input features. The predictions from individual trees are combined to produce the final prediction using averaging or voting.
5) DNGP leverages a DNN model [197] as the ranking model. The DNN model can learn to rank test inputs based on their mutation features. After training, the DNN model can generate a score that reflects their misclassification probability. This score can then be used to rank test inputs in a test set.

Compared to the mutation features of PRIMA, the distinctive aspect of GraphPrior's mutation features lies in their utilized mutation rules, which are specifically designed for GNNs. These mutation rules have the potential to directly or indirectly impact the message passing mechanism between nodes in graph data. Our experiment results in Section 6.5 demonstrate the effectiveness of the feature-based GraphPrior approaches. The observed effectiveness can be attributed, in part, to the selection of mutation rules and ranking models. Specifically, our mutation rules have been designed to generate informative mutation features by changing the massage passing between nodes in the GNN prediction process. Furthermore, our ranking models are able to utilize these mutation features for test prioritization effectively. After sufficient training, ranking models can output a misclassification score that indicates how likely a sample would be misclassified based on its mutation features. A score closer to 1 indicates a higher probability of misclassification. By sorting the misclassification scores of test inputs in descending order, the feature-based GraphPrior approaches can effectively prioritize tests that are more likely to be misclassified.

5.3.5 Usage of GraphPrior

By utilizing ranking models, GraphPrior predicts a misclassification score for each test input within a given test set. These predicted scores are then utilized for test prioritization, whereby test inputs with higher scores are prioritized higher. Particularly, the ranking models are pre-trained before the execution of GraphPrior. The training process is standardized across all the different ranking models and follows a consistent set of procedures, which are presented in detail below.
❶ **Splitting datasets** Given a GNN model G with dataset T. First, we split

the dataset T into two partitions: the training set R and the test set, in a 7:3 ratio [111]. The test set remains untouched for the purpose of evaluating GraphPrior.

❷ **Constructing the training set for ranking models** Based on the training set R, we aim to build a training set R' for training the ranking models. First, we generate a group of mutated models for each input $r_i \in R$. Then, we obtain the mutation feature vector V_i of r_i (i.e., a one-dimensional vector in which the i_{th} element denotes whether the i_{th} mutated model is killed by this input). The mutation feature vector of r_i is used to build the training set R' (i.e., the training set of the ranking models). Second, we let the original GNN model G classify each input $r_i \in R$ and compare it with the ground truth of r_i. In this way, we can identify whether r_i is misclassified by the GNN model G. If r_i is misclassified by G, we label it as 1. Otherwise, we label it as 0. In this way, we have built the ranking model training set R'.

❸ **Training ranking models** Based on R', we train the ranking models. Upon the completion of the training process, the ranking model is capable of receiving the mutation feature vector of a test input as an input and producing a misclassification score as an output. This score serves as an indicator of the probability of the test input being incorrectly classified by the GNN model.

It is worth noting that the original labels of the training set R' are binary (i.e., 1 or 0), but the ranking models that are well trained can output values (i.e., the misclassification scores). To achieve this, we make some adaptations to implement the adopted ranking algorithms (e.g., random forest and XGBoost). First, although the ranking algorithms we adopted initially deal with classification tasks, an intermediate value is calculated for the classifications. For example, if the intermediate value exceeds 0.5 (default value which can be adjusted), input will be classified into the first category; otherwise, the other category. Here, after training, we let the ranking models directly output the intermediate value, as this value can indicate the likelihood of a test input being misclassified by the GNN model, where a higher value implies a greater likelihood of misclassification. We call this intermediate value "misclassification scores" and leverage the scores of test inputs to rank them.

5.4 Study design

5.4.1 Research Questions

Our experimental evaluation answers the research questions below.

- **RQ1: How does the killing-based GraphPrior approach perform in prioritizing test inputs for GNNs?**
  In terms of test prioritization for GNNs, existing prioritization approaches usually do not take into account the interdependencies between nodes (tests) in a graph (test set). To fill the gap, we propose GraphPrior, which contains six GNN-oriented test prioritization approaches. Among them, KMGP is a killing-based approach, which regards a test input that kills more mutants as more likely to be misclassified. In this research question, we evaluate the effectiveness of the killing-based KMGP by comparing it with existing approaches that have been demonstrated as effective in detecting possibly-misclassified test inputs.
- **RQ2: How do the feature-based GraphPrior approaches perform in**

GNN test prioritization?
In addition to the killing-based KMGP, GraphPrior involves five feature-based approaches. The core difference is that, the killing-based approach regards the importance of each mutated model as equal, while the feature-based approaches learn different importance for each mutated model for test prioritization. More specifically, feature-based approaches extract features from mutation results and adopt ranking models [14, 112, 69] to utilize the mutation features for test prioritization. In this research question, we compare the effectiveness of killing-based and feature-based approaches to investigate the effect of ranking models in leveraging mutation results.

- **RQ3: How does GraphPrior perform on test inputs generated from graph adversarial attacks?**
  When faced with graph adversarial attacks, confidence-based test prioritization approaches may be fooled, thus becoming more confident in incorrect predictions. Therefore, we evaluate to what extent the effectiveness of GraphPrior is affected by graph adversarial attacks. We compare GraphPrior and confidence-based approaches [3, 47] on test inputs generated from graph adversarial attacks of existing studies [173, 174, 175, 176] to demonstrate its effectiveness.
- **RQ4: How does GraphPrior perform against different levels of graph adversarial attacks?**
  In this research question, we investigate the effectiveness of GraphPrior against different levels of graph adversarial attacks. To answer this research question, we set different levels of attacks to generate test inputs and compare GraphPrior with existing approaches to demonstrate its effectiveness.
- **RQ5: Which mutation rules generate more top contributing GNN mutants?**
  We investigate the contributions of each mutation rule in generating effective mutants of GNNs. For each GNN model, we select the top contributing mutation features to it through the XGBoost ranking algorithm [8], which is an optimized ML algorithm for ranking tasks based on the implementation of gradient boosting. We match each selected feature with the corresponding GNN mutant and identify the mutation rule that generates it. In this way, we obtain which mutation rules generate more top contributing mutants for test prioritization.
- **RQ6: Can GraphPrior and the uncertainty-based metrics be used in active learning scenarios to improve a GNN model by retraining?**
  In the face of a large number of unlabeled inputs and a limited time budget, it is not feasible to manually label all the inputs and use them to retrain a GNN. One established solution to reduce data labeling costs is active learning [198], which involves selecting informative subsets of training samples to improve the model performance. In this research question, we investigate the effectiveness of GraphPrior and the uncertainty-based metrics in selecting informative retraining inputs to improve the quality of a GNN model.

5.4.2 GNN models and Datasets

In our study, we totally adopt 604 subjects to evaluate the effectiveness of GraphPrior and the compared approaches [3, 47]. Table 5.1 exhibits their basic information. Among the 604 subjects considered in this study, 16 subjects were utilized in the experiments of RQ1, 16 subjects in RQ2, 108 subjects in RQ3, 432

subjects in RQ4, 16 subjects in RQ5 and 16 subjects in RQ6. It is worth noting that, among these subjects, a total of 64 subjects (which were utilized in RQ1, RQ5, and RQ6) were associated with clean datasets, while the remaining 540 subjects (which were utilized in RQ3 and RQ4) were associated with adversarial datasets.

Our study involves four GNN models: GCN (Graph Convolutional Networks) [163], GAT (Graph Attention Networks) [185], GraphSAGE (Graph SAmple and aggreGatE) [191] and TAGCN (Topology Adaptive Graph Convolutional Network) [192], tested by four datasets, namely the Cora [186], CiteSeer [186], PubMed [186] and LastFM [187]. We present their descriptions as follows.

5.4.2.1 GNN Models

- **GCN** [163] GCN is a class of convolutional neural networks that can work directly on the graph. It solves the problem of classifying nodes (such as documents) in graphs (such as citation networks), of which only a small number of nodes are labeled. The core idea of GCN is to use the edge information of a graph to aggregate node information to generate new node representations. GCN has been used in several existing studies [199, 200, 201].
- **GAT** [185] GAT introduces a self-attention mechanism in the propagation process. Compared to GCN, which regards all neighbors of a node equally, the attention mechanism assigns different attention scores to each neighbor, thereby identifying more important neighbors.
- **GraphSAGE** [191] GraphSAGE is a generalized inductive framework that generates node embeddings by sampling and aggregating features of neighbor nodes.
- **TAGCN** [192] TAGCN introduces a systematic approach to design a set of fixed-size learnable filters to perform convolutions on graphs. These filters are topology-fit to the topology of the graph as they scan the graph for convolution.

5.4.2.2 Datasets

- **Cora** [186] The Cora dataset is a citation graph composed of 2,708 scientific publications (nodes) and 5,429 links (edges) between them. Nodes represent ML papers, and edges represent citations between pairs of papers. Each paper is classified into one of seven classes, such as reinforcement learning and neural networks.
- **CiteSeer** [186] The CiteSeer dataset consists of 3,327 scientific publications (nodes) and 4,732 links (edges). Each paper belongs to one of six categories such as AI and ML.
- **PubMed** [186] The PubMed dataset contains 19,717 diabetes-related scientific publications (nodes) and 44,338 links (edges). Publications are classified into three classes such as Cancer and AIDS (i.e., Acquired Immune Deficiency Syndrome).
- **LastFM Asia Social Network** [187] The dataset LastFM Asia Social Network was collected from the social network of users on the Last.fm music platform in Asia. Nodes are LastFM users, and edges are mutual follower relationships between them. LastFM contains 7,624 nodes and 27,806 edges. The classification task of the LastFM dataset is to predict the home country of a user (e.g., Philippines, Malaysia, Singapore).

Notably, we evaluate GraphPrior on different types of test inputs (i.e., both natural test inputs and adversarial test inputs. We adopted eight graph adversarial

Table 5.1: GNN models and datasets

ID	Dataset	#Nodes	#Edges	Model	Type
1	CiteSeer	3327	4732	GCN	Original, DICE, MMA, PGD, RAA, RAF, RAR
2	CiteSeer	3327	4732	GAT	Original, DICE, MMA, PGD, RAA, RAF, RAR
3	CiteSeer	3327	4732	TAGCN	Original, DICE, MMA, PGD, RAA, RAF, RAR
4	CiteSeer	3327	4732	GraphSAGE	Original, DICE, MMA, PGD, RAA, RAF, RAR
5	Cora	2708	5429	GCN	Original, DICE, MMA, PGD, RAA, RAF, RAR, NEAR, NEAA
6	Cora	2708	5429	GAT	Original, DICE, MMA, PGD, RAA, RAF, RAR, NEAR, NEAA
7	Cora	2708	5429	TAGCN	Original, DICE, MMA, PGD, RAA, RAF, RAR, NEAR, NEAA
8	Cora	2708	5429	GraphSAGE	Original, DICE, MMA, PGD, RAA, RAF, RAR, NEAR, NEAA
9	LastFM	7624	27806	GCN	Original, DICE, PGD, RAA, RAF, RAR, NEAR, NEAA
10	LastFM	7624	27806	GAT	Original, DICE, PGD, RAA, RAF, RAR, NEAR, NEAA
11	LastFM	7624	27806	TAGCN	Original, DICE, PGD, RAA, RAF, RAR, NEAR, NEAA
12	LastFM	7624	27806	GraphSAGE	Original, DICE, PGD, RAA, RAF, RAR, NEAR, NEAA
13	PubMed	19717	44338	GCN	Original, DICE, RAA, RAF, RAR, NEAR, NEAA
14	PubMed	19717	44338	GAT	Original, DICE, RAA, RAF, RAR, NEAR, NEAA
15	PubMed	19717	44338	TAGCN	Original, DICE, RAA, RAF, RAR, NEAR, NEAA
16	PubMed	19717	44338	GraphSAGE	Original, DICE, RAA, RAF, RAR, NEAR, NEAA

attacks, presented in Section 5.4.4.

5.4.3 Compared Approaches

In our study, we considered 7 compared approaches in total, including one baseline (i.e., random selection), four DNN test prioritization approaches and two active learning approaches. We select these approaches due to the following reasons: 1) These approaches can be adapted for GNN test prioritization; 2) The selected approaches have been demonstrated as effective for DNNs in existing studies [3] [47] [7]; 3) The implementations of these approaches have been released by the authors.

- **DeepGini** DeepGini [3] prioritizes test inputs based on model confidence. DeepGini leverages the Gini coefficient to measure the likelihood of a test input being misclassified. DeepGini leverages Formula 5.1 to calculate the ranking scores.

$$\xi(x) = 1 - \sum_{i=1}^{N} \left(p_i(x)\right)^2 \tag{5.1}$$

  where $\xi(x)$ refers to the likelihood of the test input x being misclassified. $p_i(x)$ refers to the probability that the test input x is predicted to be label i. N refers to the number of labels.
- **Margin** Margin [172] regards a test input with less difference between the top two most confidence predictions as more likely to be misclassified. Margin score is calculated by Formula 5.2.

$$M(x) = p_k(x) - p_j(x) \tag{5.2}$$

  where $M(x)$ refers to the margin score. $p_k(x)$ refers to the most confident prediction probability. $p_j(x)$ refers to the second most confident prediction probability.
- **Least Confidence** Least Confidence [172] regards test inputs for which the model has the least confidence as more likely to be misclassified. Least confidence is calculated by Formula 5.3.

$$L(x) = \max_{i=1:n} p_i(x) \tag{5.3}$$

  where $L(x)$ refers to the confidence score. $p_i(x)$ refers to the probability that the test input x is predicted to be label i via a model M.
- **Vanilla Softmax** Vanilla Softmax [7] is computed by subtracting the highest activation probability in the output softmax layer from 1, resulting in a metric that is positively correlated with the misclassification probability. Formula 5.4

presents the calculation of the Vanilla Softmax metric.

$$\mathrm{V}(x) = 1 - \max_{c=1}^{C} l_c(x) \tag{5.4}$$

where $l_c(x)$ belongs to a valid softmax array in which all values are between 0 and 1, and their sum is 1.

- **Prediction-Confidence Score (PCS)** PCS [7] calculates the difference between the predicted class and the second most confident class in softmax likelihood.
- **Entropy** Entropy [7] calculates the entropy of the softmax likelihood.
- **Random selection** [133] In random selection, the execution order of the test inputs is determined randomly.

5.4.4 Graph Adversarial Attacks

In RQ3 and RQ4, we evaluate the effectiveness of GraphPrior on test inputs generated through diverse graph adversarial attacks, in which attackers aim to generate graph adversarial perturbations by manipulating the graph structure or node features to fool the GNN models. We introduce all the attacks we applied in our experiments as follows.

- **Disconnect Internally, Connect Externally (DICE)** [173] The DICE attack is a type of white-box attack whereby the adversary has access to all information about the targeted GNN model, including its parameters, training data, labels, and predictions. Specifically, the DICE attack randomly adds edges between nodes with different labels or removes edges between nodes sharing the same label. Through this, the attack can generate adversarial perturbations that can fool the targeted GNN model.
- **PGD attack** [174] The PGD attack leverages the Projected Gradient Descent (PGD) algorithm to search for optimal structural perturbations to attack GNNs.
- **Min-max attack (MMA)** [174] The min-max attack is a type of untargeted white-box GNN attack. The attack problem is formulated as a min-max problem, where the inner maximization is designed to update the model's parameters (θ) by maximizing the attack loss, and it can be solved using gradient ascent. On the other hand, the outer minimization can be achieved by using Projected Gradient Descent (PGD) [202].
- **Node Embedding Attack-Add (NEAA)** [175] In node embedding attack-add, the attackers are capable of modifying the original graph structure by adding new edges while adhering to a predefined budget constraint.
- **Node Embedding Attack-Remove (NEAR)** [175] In node embedding attack-remove, the attackers modify the original graph structure by removing edges.
- **Random Attack-Add (RAA)** [176] The Random Attack-Add approach randomly adds edges to the input graph to fool the targeted GNN model.
- **Random Attack-Flip (RAF)** [176] The Random Attack-Flip approach randomly flips edges to the input graph to fool the targeted GNN model.
- **Random Attack-Remove (RAR)** [176] The Random Attack-Add approach randomly removes edges to the input graph to fool the targeted GNN model.

5.4.5 Evaluation of mutation rules (RQ5)

In RQ5, we investigated the contribution of different mutation rules in generating top contributing mutated models. First, for each GNN model, we utilize the cover metric in XGBoost [8] to evaluate the importance of its mutation features and rank

them according to the descending order of the importance scores. The cover metric can evaluate the importance of mutation features by quantifying the average coverage of each instance by the leaf nodes in a decision tree. Specifically, it calculates the number of times a particular feature is used to split the data across all trees in the ensemble and then sums up the coverage values for each feature over all trees. This coverage value is then normalized by the total number of instances to obtain the average coverage of each instance by the leaf nodes. The importance of a feature is then calculated based on its coverage value, and features with higher coverage values are considered more important.

Upon obtaining the importance of each mutation feature, which corresponds to a specific mutated model, we proceed to match and determine the importance of the respective mutated models. Subsequently, we select the top N critical mutated models and identify the specific mutated rules employed in their generation. This enables a comparative analysis of the contributions of various mutation rules.

5.4.6 Implementation and Configuration

We implemented GraphPrior in Python based on the PyTorch 1.11.0 framework [203]. We also integrate the available implementations of the compared approaches [7, 3, 46, 172] into our experimental pipeline to adapt to the GNN prioritization problem. Regarding our mutation rules, we set the number of mutated models as 80~240 across different subjects. Balancing the trade-off between execution time and the effectiveness of GraphPrior is a critical consideration in determining the number of mutants. Building on relevant literature [2], we identified a suitable range of mutants. Our preliminary investigations on multiple subjects demonstrate that these settings effectively maintain the effectiveness of GraphPrior while controlling the runtime within a reasonable range. In the case of subjects associated with longer mutant generation times, we choose to generate a comparatively smaller number of mutants compared to other subjects. Additionally, the range was achieved through the full execution of all pre-defined mutation rules. It is worth noting that the total number of mutation rules was predetermined and fixed. Thus, even with the addition of new mutants, the impact on the performancethe trade-off between excessive computational time and the preservation of method effectiveness of GraphPrior is minor, as the new mutants are created based on the existing mutation rules.

With regard to the specific mutation rules that change the integer/float training parameters, we define a parameter range close to the original parameter values, in order to achieve slight mutations. We conducted a preliminary study using multiple subjects, demonstrating the effectiveness of such settings. Moreover, to obtain parameter values from the specified range, we adopt uniform sampling [48] as the sampling methodology. This technique ensures an equitable probability of selecting each value within the parameter range and has been widely adopted across the ML testing field [48, 204, 205].

More specifically, we set the hidden channel parameter in the range of [15-20), epochs parameter as <= 50, heads parameter as <= 5, and negative slope parameter as <= 0.2. For the mutation rules that change the Boolean type parameters, if the parameter value of the original model is true, we set it to false. If the original value is false, we set it to true. The parameter ranges for our mutation rules are carefully selected to ensure the change to the original GNN model is slight.

With respect to the configuration of the ranking models utilized in GraphPrior, we

made several parameter selections: for the random forest, XGBoost, and LightGBM ranking algorithms, we set the n_estimators parameter to 100. For the DNN ranking model, we set the learning_rate parameter to 0.01. Finally, for the logistic regression ranking algorithm, we set the max_iter parameter to 100.

We conducted the following experiments on a high-performance computer cluster, and each cluster node runs a 2.6 GHz Intel Xeon Gold 6132 CPU with an NVIDIA Tesla V100 16G SXM2 GPU. For the data process, we conducted corresponding experiments on a MacBook Pro laptop with Mac OS Big Sur 11.6, Intel Core i9 CPU, and 64 GB RAM.

5.4.7 Measurements

Following the existing study [3], we leverage Average Percentage of Fault-Detection (APFD) [92] to evaluate the prioritization effectiveness of GraphPrior and the compared approaches. APFD is a standard metric for prioritization evaluation. Typically, higher APFD values indicate faster misclassification detection rates. We calculate the APFD values by Formula 5.5

$$APFD = 1 - \frac{\sum_{i=1}^{k} o_i}{kn} + \frac{1}{2n} \tag{5.5}$$

where n is the number of test inputs in the test set T. k is the number of test inputs in T that will be misclassified by the GNN model G. o_i is the index of the i_{th} misclassified tests in the prioritized test set. More specifically, o_i is an integer that represents the position of the i_{th} misclassified tests in the test set that has been prioritized. When $\sum_{i=1}^{k} o_i$ is small (i.e., the total index sum of the misclassified tests within the prioritized list is small), indicating that that the misclassified tests are prioritized higher, the APFD will be large according to Formula 5.5. Therefore, large APFD indicates better prioritization effectiveness. Following the existing study [3], we normalize the APFD values to [0,1]. We consider a prioritization approach better when the APFD value is closer to 1. We present the comparison results in tables.

For more detailed analysis, we utilize PFD (Percentage of Fault Detected) [3] to evaluate the fault detection rate of each approach on different ratios of prioritized test inputs. High PFD values refer to high effectiveness in detecting misclassified test inputs.

$$PFD = \frac{F_c}{F_t} \tag{5.6}$$

where F_c is the number of faults (i.e., misclassified test inputs) correctly detected. F_t is the total number of faults. More specifically, we evaluate the fault detection rate of GraphPrior against different ratios of prioritized tests. We use **PFD-n** to represent the first n% prioritized test inputs.

5.5 Results and analysis

5.5.1 RQ1: Effectiveness of the killing-based GraphPrior approach (KMGP)

Objectives: We investigate the effectiveness of the killing-based GraphPrior approach, KMGP (cf. Section 5.3.3), comparing it with existing approaches that can be used to identify possibly-misclassified test inputs.

Experimental design: We used 16 pairs of datasets and GNN models as subjects to evaluate the effectiveness of GraphPrior. Table 5.1 exhibits their basic information. We carefully selected 7 compared approaches (i.e., DeepGini, least confidence, margin, Vanilla SM, PCS, entropy, and random selection), which can be adapted for GNN test prioritization. Random selection is considered the baseline. We adopt two metrics to measure the effectiveness of GraphPrior and the compared approaches: Average Percentage of Fault-Detection (APFD) and Percentage of Fault Detected (PFD), which are explained in Section 5.4.7.

Due to the randomness of the training process of a GNN model, we conduct a statistical analysis by repeating all the experiments 10 times. More specifically, for each subject (a dataset with a GNN model), 10 different GNN models are generated through separate training processes.

Results: The GraphPrior approach KMGP outperforms all the compared approaches (i.e., DeepGini, Least Confidence, Margin, Vanilla SM, PCS, Entropy, and Random) in GNN test prioritization. Table 5.2 presents the comparison results of the killing-based GraphPrior approach (KMGP) and a set of compared approaches using the APFD metric. We highlight the approach with the highest effectiveness for each case in grey. The results demonstrate that KMGP outperforms the other approaches in the majority of cases, specifically in 87.5% (14 out of 16) subjects. Vanilla SM, on the other hand, performs the best in only 12.5% of cases. Additionally, the average APFD value achieved by KMGP was 0.748, which is higher than that of the compared techniques, with improvements of 4.76%~49.6%. These results suggest that KMGP offers a promising solution for prioritizing GNN test inputs.

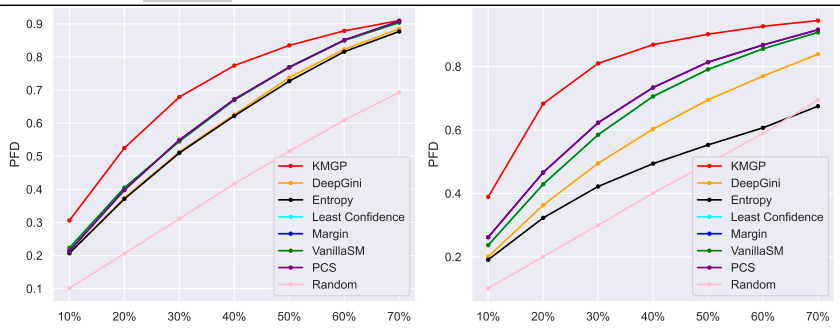
Table 5.3 exhibits the comparison results among the test prioritization techniques with respect to PFD. We highlight the approach with the highest effectiveness for each case in grey. The findings indicate that, for 68.75% (11 out of 16) of the subjects, KMGP performs best when prioritizing less than 50% of tests. Furthermore, for a majority of the subjects, specifically 87.5% (14 out of 16), KMGP exhibits the best performance when prioritizing less than 30% of tests. Furthermore, Table 5.4 exhibits the overall comparison results in terms of PFD. We can see that when prioritizing 10%~30% test inputs, the average effectiveness of KMGP outperforms that of the compared approaches in 100% cases. When prioritizing 10%~50% test inputs, the average effectiveness of KMGP outperforms that of the compared approaches in 90% cases. Figure 5.2 plots the ratio of detected misclassified tests against the prioritized tests. We see that GraphPrior achieves a higher APFD value in comparison to DeepGini, entropy, least confidence, margin, Vanilla SM, PCS, and random. These results confirm the effectiveness of KMGP in GNN test input prioritization.

To demonstrate the stability of our findings, a statistical analysis is performed. Specifically, all the experiments are repeated ten times for each subject, resulting in 10 distinct GNN model instances obtained through separate training processes for a given original GNN model. Based on the statistical analysis of the resulting data, the p-value was found to be lower than 10^{-05}, indicating that the KMGP approach can consistently outperform the compared approaches in terms of test prioritization.

> **Answer to RQ1:** *The GraphPrior approach KMGP outperforms all the compared approaches (i.e., DeepGini, Least Confidence, Margin, Vanilla SM, PCS, Entropy and Random) in GNN test prioritization.*

Table 5.2: Effectiveness comparison among KMGP and the compared approaches in terms of APFD

Data	Model	Approaches							
		KMGP	DeepGini	Least Confidence	Margin	Vanilla SM	PCS	Entropy	Random
CiteSeer	GAT	0.708	0.671	0.691	0.694	0.691	0.694	0.646	0.508
	GCN	0.701	0.641	0.677	0.682	0.677	0.682	0.638	0.502
	GraphSAGE	0.739	0.663	0.684	0.684	0.684	0.684	0.659	0.497
	TAGCN	0.712	0.658	0.691	0.694	0.691	0.694	0.620	0.499
Cora	GAT	0.841	0.742	0.770	0.763	0.770	0.763	0.733	0.487
	GCN	0.812	0.690	0.736	0.739	0.736	0.739	0.684	0.495
	GraphSAGE	0.792	0.727	0.781	0.784	0.781	0.784	0.704	0.515
	TAGCN	0.782	0.701	0.739	0.738	0.739	0.738	0.690	0.498
LastFM	GAT	0.801	0.633	0.695	0.713	0.695	0.713	0.534	0.498
	GCN	0.761	0.713	0.758	0.746	0.758	0.746	0.603	0.497
	GraphSAGE	0.702	0.734	0.761	0.754	0.761	0.754	0.626	0.502
	TAGCN	0.673	0.719	0.741	0.730	0.741	0.730	0.657	0.498
PubMed	GAT	0.735	0.642	0.670	0.661	0.670	0.661	0.645	0.502
	GCN	0.748	0.645	0.680	0.670	0.680	0.670	0.647	0.501
	GraphSAGE	0.747	0.631	0.685	0.675	0.685	0.675	0.634	0.498
	TAGCN	0.720	0.613	0.663	0.672	0.663	0.672	0.615	0.497
Average		0.748	0.677	0.714	0.712	0.714	0.712	0.646	0.500



a) CiteSeer, GraphSAGE **b)** LastFM, GAT]

Figure 5.2: Test prioritization effectiveness among KMGP and the compared approaches for CiteSeer with GraphSAGE and LastFM with GAT. X-Axis: the percentage of prioritized tests; Y-Axis: the percentage of detected miscalssified tests.

5.5.2 RQ2: Effectiveness of the feature-based GraphPrior approaches

Objectives: We investigate the effectiveness of feature-based approaches in GraphPrior, including XGGP, LRGP, RFGP, LGGP, and DNGP, compared with the killing-based approach KMGP.

Experimental design: We evaluated the effectiveness of feature-based GraphPrior approaches with the killing-based approach KMGP on 16 subjects (four graph datasets × four GNN models). Due to the randomness of the training process of a GNN model, we repeat all the experiments ten times and calculate the average results. For each subject (a dataset with a GNN model), 10 different GNN models are generated through separate training processes. For evaluation, we calculated the APFD (Average Percentage of Fault-Detection) values of all the approaches on each subject, which can reflect the misclassification detection rate. Moreover, we calculated the PFD (Percentage of Fault Detected) values of all the approaches on different ratios of prioritized tests to further investigate the effectiveness of feature-based approaches.

Results: The experimental results of this research question are exhibited in Table 5.5, Table 5.6 and Table 5.7. Table 5.5 presents the comparison results in terms of APFD, while Table 5.6 and Table 5.7 present the comparison results in terms of PFD.

Table 5.3: Effectiveness comparison among KMGP and the compared approaches in terms of PFD

Data	Model	Approaches	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60	PFD-70
CiteSeer	GAT	**KMGP**	0.264	0.464	0.629	0.750	0.812	0.841	0.875
		DeepGini	0.211	0.382	0.521	0.646	0.748	0.828	0.895
		Entropy	0.203	0.373	0.506	0.621	0.716	0.788	0.844
		Least Confidence	0.231	0.409	0.550	0.680	0.777	0.861	0.913
		Margin	0.228	0.401	0.547	0.688	0.794	0.864	0.914
		Vanilla SM	0.231	0.409	0.550	0.680	0.777	0.861	0.913
		PCS	0.228	0.401	0.547	0.688	0.794	0.864	0.914
		Random	0.099	0.192	0.296	0.391	0.493	0.591	0.689
	GCN	**KMGP**	0.278	0.492	0.652	0.723	0.771	0.811	0.865
		DeepGini	0.200	0.355	0.490	0.600	0.697	0.783	0.858
		Entropy	0.201	0.354	0.487	0.595	0.692	0.779	0.856
		Least Confidence	0.229	0.406	0.544	0.661	0.748	0.827	0.889
		Margin	0.214	0.399	0.556	0.674	0.776	0.844	0.895
		Vanilla SM	0.229	0.406	0.544	0.661	0.748	0.827	0.889
		PCS	0.214	0.399	0.556	0.674	0.776	0.844	0.895
		Random	0.098	0.197	0.292	0.388	0.488	0.587	0.690
	GraphSAGE	**KMGP**	0.306	0.525	0.679	0.774	0.835	0.879	0.910
		DeepGini	0.208	0.374	0.513	0.626	0.738	0.823	0.885
		Entropy	0.207	0.371	0.510	0.622	0.727	0.816	0.877
		Least Confidence	0.223	0.405	0.545	0.670	0.769	0.850	0.904
		Margin	0.214	0.398	0.549	0.672	0.769	0.851	0.908
		Vanilla SM	0.223	0.405	0.545	0.670	0.769	0.850	0.904
		PCS	0.214	0.398	0.549	0.672	0.769	0.851	0.908
		Random	0.101	0.206	0.311	0.417	0.515	0.609	0.693
	TAGCN	**KMGP**	0.295	0.490	0.617	0.723	0.795	0.845	0.888
		DeepGini	0.216	0.375	0.512	0.622	0.719	0.808	0.877
		Entropy	0.214	0.366	0.492	0.592	0.693	0.749	0.801
		Least Confidence	0.246	0.427	0.570	0.678	0.772	0.845	0.905
		Margin	0.234	0.430	0.578	0.688	0.776	0.850	0.907
		Vanilla SM	0.246	0.427	0.570	0.678	0.772	0.845	0.905
		PCS	0.234	0.430	0.578	0.688	0.776	0.850	0.907
		Random	0.101	0.196	0.297	0.383	0.482	0.586	0.684
LastFM	GAT	**KMGP**	0.389	0.683	0.810	0.869	0.902	0.927	0.945
		DeepGini	0.201	0.363	0.495	0.603	0.695	0.770	0.839
		Entropy	0.191	0.323	0.422	0.494	0.553	0.607	0.675
		Least Confidence	0.237	0.429	0.585	0.706	0.791	0.856	0.908
		Margin	0.262	0.466	0.623	0.734	0.814	0.868	0.916
		Vanilla SM	0.237	0.429	0.585	0.706	0.791	0.856	0.908
		PCS	0.262	0.466	0.623	0.734	0.814	0.868	0.916
		Random	0.101	0.201	0.300	0.401	0.495	0.589	0.695
	GCN	**KMGP**	0.403	0.648	0.728	0.770	0.830	0.868	0.915
		DeepGini	0.267	0.467	0.600	0.715	0.799	0.875	0.928
		Entropy	0.254	0.411	0.501	0.570	0.627	0.685	0.755
		Least Confidence	0.298	0.530	0.691	0.799	0.880	0.927	0.956
		Margin	0.278	0.499	0.661	0.783	0.865	0.920	0.951
		Vanilla SM	0.298	0.530	0.691	0.799	0.880	0.927	0.956
		PCS	0.278	0.499	0.661	0.783	0.865	0.920	0.951
		Random	0.098	0.199	0.302	0.397	0.503	0.600	0.704
	GraphSAGE	**KMGP**	0.302	0.482	0.580	0.668	0.800	0.842	0.902
		DeepGini	0.285	0.501	0.655	0.765	0.836	0.893	0.929
		Entropy	0.283	0.443	0.520	0.587	0.649	0.708	0.775
		Least Confidence	0.294	0.527	0.709	0.825	0.892	0.922	0.946
		Margin	0.276	0.525	0.700	0.819	0.883	0.913	0.944
		Vanilla SM	0.294	0.527	0.709	0.825	0.892	0.922	0.946
		PCS	0.276	0.525	0.700	0.819	0.883	0.913	0.944
		Random	0.095	0.194	0.298	0.398	0.498	0.596	0.697
	TAGCN	**KMGP**	0.250	0.431	0.544	0.644	0.706	0.819	0.892
		DeepGini	0.260	0.461	0.615	0.731	0.821	0.885	0.934
		Entropy	0.258	0.451	0.577	0.653	0.720	0.769	0.816
		Least Confidence	0.260	0.475	0.642	0.769	0.865	0.928	0.966
		Margin	0.238	0.450	0.616	0.755	0.856	0.922	0.962
		Vanilla SM	0.260	0.475	0.642	0.769	0.865	0.928	0.966
		PCS	0.238	0.450	0.616	0.755	0.856	0.922	0.962
		Random	0.100	0.203	0.299	0.401	0.497	0.596	0.697
Cora	GAT	**KMGP**	0.454	0.759	0.884	0.919	0.939	0.954	0.975
		DeepGini	0.295	0.509	0.669	0.781	0.852	0.892	0.928
		Entropy	0.293	0.503	0.658	0.766	0.842	0.886	0.918
		Least Confidence	0.296	0.539	0.724	0.830	0.899	0.932	0.962
		Margin	0.282	0.525	0.708	0.815	0.879	0.934	0.970
		Vanilla SM	0.296	0.539	0.724	0.830	0.899	0.932	0.962
		PCS	0.282	0.525	0.708	0.815	0.879	0.934	0.970
		Random	0.099	0.192	0.294	0.392	0.478	0.578	0.679
	GCN	**KMGP**	0.384	0.704	0.854	0.884	0.909	0.933	0.952
		DeepGini	0.249	0.418	0.569	0.682	0.776	0.853	0.908
		Entropy	0.245	0.411	0.559	0.676	0.763	0.840	0.897
		Least Confidence	0.265	0.480	0.643	0.770	0.848	0.906	0.954
		Margin	0.254	0.469	0.653	0.781	0.860	0.912	0.956
		Vanilla SM	0.265	0.480	0.643	0.770	0.848	0.906	0.954
		PCS	0.254	0.469	0.653	0.781	0.860	0.912	0.956
		Random	0.097	0.197	0.291	0.398	0.505	0.596	0.695
	GraphSAGE	**KMGP**	0.489	0.705	0.777	0.820	0.848	0.886	0.919
		DeepGini	0.323	0.498	0.623	0.736	0.829	0.878	0.922
		Entropy	0.318	0.482	0.604	0.710	0.792	0.846	0.885
		Least Confidence	0.356	0.584	0.723	0.833	0.903	0.940	0.962
		Margin	0.363	0.604	0.735	0.830	0.897	0.939	0.964
		Vanilla SM	0.356	0.584	0.723	0.833	0.903	0.940	0.962
		PCS	0.363	0.604	0.735	0.830	0.897	0.939	0.964
		Random	0.107	0.205	0.306	0.403	0.500	0.596	0.691
	TAGCN	**KMGP**	0.372	0.668	0.788	0.841	0.863	0.888	0.914
		DeepGini	0.249	0.450	0.586	0.696	0.783	0.857	0.914
		Entropy	0.246	0.442	0.578	0.689	0.771	0.838	0.895
		Least Confidence	0.273	0.481	0.638	0.762	0.850	0.913	0.954
		Margin	0.255	0.466	0.638	0.764	0.861	0.922	0.964
		Vanilla SM	0.273	0.481	0.638	0.762	0.850	0.913	0.954
		PCS	0.255	0.466	0.638	0.764	0.861	0.922	0.964
		Random	0.102	0.204	0.309	0.403	0.507	0.592	0.699
PubMed	GAT	**KMGP**	0.336	0.588	0.697	0.754	0.813	0.859	0.893
		DeepGini	0.205	0.359	0.495	0.607	0.702	0.782	0.856
		Entropy	0.205	0.360	0.496	0.609	0.707	0.788	0.864
		Least Confidence	0.213	0.384	0.532	0.657	0.758	0.841	0.895
		Margin	0.215	0.388	0.532	0.656	0.750	0.817	0.871
		Vanilla SM	0.213	0.384	0.532	0.657	0.758	0.841	0.895
		PCS	0.215	0.388	0.532	0.656	0.750	0.817	0.871
		Random	0.101	0.201	0.298	0.396	0.497	0.595	0.696
	GCN	**KMGP**	0.347	0.607	0.743	0.788	0.826	0.860	0.894
		DeepGini	0.215	0.395	0.534	0.624	0.698	0.771	0.838
		Entropy	0.216	0.395	0.535	0.626	0.701	0.774	0.842
		Least Confidence	0.223	0.407	0.560	0.686	0.782	0.844	0.890
		Margin	0.211	0.397	0.550	0.679	0.768	0.832	0.876
		Vanilla SM	0.223	0.407	0.560	0.686	0.782	0.844	0.890
		PCS	0.211	0.397	0.550	0.679	0.768	0.832	0.876
		Random	0.098	0.202	0.302	0.403	0.503	0.602	0.704
	GraphSAGE	**KMGP**	0.396	0.635	0.713	0.757	0.808	0.850	0.889
		DeepGini	0.214	0.364	0.488	0.589	0.676	0.756	0.829
		Entropy	0.215	0.365	0.490	0.591	0.680	0.761	0.834
		Least Confidence	0.229	0.407	0.561	0.682	0.774	0.846	0.901
		Margin	0.229	0.412	0.555	0.668	0.756	0.832	0.884
		Vanilla SM	0.229	0.407	0.561	0.682	0.774	0.846	0.901
		PCS	0.229	0.412	0.555	0.668	0.756	0.832	0.884
		Random	0.096	0.200	0.303	0.400	0.505	0.606	0.704
	TAGCN	**KMGP**	0.379	0.545	0.610	0.722	0.791	0.844	0.885
		DeepGini	0.210	0.352	0.468	0.553	0.644	0.732	0.811
		Entropy	0.211	0.354	0.470	0.557	0.650	0.736	0.814
		Least Confidence	0.223	0.397	0.541	0.658	0.744	0.815	0.867
		Margin	0.232	0.414	0.566	0.675	0.761	0.822	0.868
		Vanilla SM	0.223	0.397	0.541	0.658	0.744	0.815	0.867
		PCS	0.232	0.414	0.566	0.675	0.761	0.822	0.868
		Random	0.099	0.196	0.297	0.398	0.499	0.601	0.700

Table 5.4: Average comparison results among KMGP and the compared approaches in terms of PFD

Data	Approaches	#Best case in PFD							Average PFD						
		PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60	PFD-70	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60	PFD-70
CiteSeer	**KMGP**	4	4	4	4	3	1	1	0.285	0.492	0.644	0.742	0.803	0.844	0.884
	DeepGini	0	0	0	0	0	0	0	0.208	0.371	0.509	0.623	0.725	0.810	0.878
	Entropy	0	0	0	0	0	0	0	0.206	0.366	0.498	0.607	0.707	0.783	0.844
	Least Confidence	0	0	0	0	0	0	0	0.232	0.411	0.552	0.672	0.766	0.845	0.902
	Margin	0	0	0	0	0	0	0	0.222	0.407	0.557	0.680	0.778	0.852	0.906
	Vanilla SM	0	0	0	0	0	0	0	0.232	0.411	0.552	0.672	0.766	0.845	0.902
	PCS	0	0	0	0	1	3	3	0.222	0.407	0.557	0.680	0.778	0.852	0.906
	Random	0	0	0	0	0	0	0	0.099	0.197	0.299	0.394	0.494	0.593	0.689
Cora	**KMGP**	4	4	4	3	3	2	1	0.424	0.709	0.825	0.866	0.889	0.915	0.940
	DeepGini	0	0	0	0	0	0	0	0.279	0.468	0.611	0.723	0.810	0.870	0.918
	Entropy	0	0	0	0	0	0	0	0.275	0.459	0.599	0.710	0.792	0.852	0.898
	Least Confidence	0	0	0	0	0	0	0	0.297	0.521	0.681	0.798	0.875	0.922	0.958
	Margin	0	0	0	0	0	0	0	0.288	0.516	0.683	0.797	0.874	0.926	0.963
	Vanilla SM	0	0	0	1	1	1	0	0.297	0.521	0.681	0.798	0.875	0.922	0.958
	PCS	0	0	0	0		1	3	0.288	0.516	0.683	0.797	0.874	0.926	0.963
	Random	0	0	0	0	0	0	0	0.101	0.199	0.300	0.399	0.497	0.590	0.691
LastFM	**KMGP**	3	2	2	1	1	1	1	0.336	0.561	0.665	0.737	0.809	0.864	0.913
	DeepGini	0	0	0	0	0	0	0	0.253	0.448	0.591	0.703	0.787	0.855	0.907
	Entropy	0	0	0	0	0	0	0	0.246	0.407	0.505	0.576	0.637	0.692	0.755
	Least Confidence	0	0	0	0	0	0	0	0.272	0.490	0.656	0.774	0.857	0.908	0.944
	Margin	0	0	0	0	0	0	0	0.263	0.485	0.650	0.772	0.854	0.905	0.943
	Vanilla SM	1	2	2	3	3	3	3	0.272	0.490	0.656	0.774	0.857	0.908	0.944
	PCS	0	0	0	0	0	0	0	0.263	0.485	0.65	0.772	0.854	0.905	0.943
	Random	0	0	0	0	0	0	0	0.098	0.199	0.299	0.399	0.498	0.595	0.698
PubMed	**KMGP**	4	4	4	4	4	4	2	0.364	0.593	0.690	0.755	0.809	0.853	0.890
	DeepGini	0	0	0	0	0	0	0	0.211	0.367	0.496	0.593	0.679	0.760	0.833
	Entropy	0	0	0	0	0	0	0	0.211	0.368	0.497	0.595	0.684	0.764	0.838
	Least Confidence	0	0	0	0	0	0	0	0.222	0.398	0.548	0.670	0.764	0.836	0.888
	Margin	0	0	0	0	0	0	2	0.221	0.402	0.550	0.669	0.758	0.825	0.874
	Vanilla SM	0	0	0	0	0	0	0	0.222	0.398	0.548	0.670	0.764	0.836	0.888
	PCS	0	0	0	0	0	0	0	0.221	0.402	0.550	0.669	0.758	0.825	0.874
	Random	0	0	0	0	0	0	0	0.098	0.199	0.300	0.399	0.501	0.601	0.701

Among all the GraphPrior approaches, RFGP demonstrates the highest level of effectiveness in most cases. Table 5.5 exhibits the comparison results among KMGP (i.e., the killing-based GraphPrior approach) and the feature-based GraphPrior approaches in terms of APFD. The results demonstrate RFGP outperforms other GraphPrior approaches on average. Moreover, the average APFD values of RFGP exceed that of KMGP by around 0.02. Additionally, across different subjects, RFGP outperforms other GraphPrior approaches in the majority of cases. To provide a more detailed analysis, Table 5.6 and Table 5.7 exhibit the comparison results of all GraphPrior approaches in terms of PFD. The findings also confirm that RFGP is the most effective GraphPrior approach. Furthermore, Table 5.7 indicates that, on average, RFGP is consistently more effective than other GraphPrior approaches across different test prioritization ratios. Figure 5.3 presents some examples aimed at providing a more visually intuitive understanding of the performance of the various GraphPrior approaches. Collectively, these results suggest that RFGP is the most effective GraphPrior approach for the evaluated datasets.

Additionally, although the killing-based GraphPrior approach, KMGP, shows good effectiveness in some specific datasets, its average effectiveness is lower than several feature-based GraphPrior approaches, such as RFGP, LGGP, and XGGP. This result suggests that KMGP is less stable compared to some feature-based approaches. For example, in Figure 5.3b), we can see that KMGP (represented by the red line) is less effective than other GraphPrior approaches. In fact, the main difference between KMGP and feature-based GraphPrior approaches lies in their strategy for utilizing mutation results. Specifically, KMGP treats all mutated models as having equal importance, whereas feature-based GraphPrior approaches, such as RFGP, employ ranking models to assign higher weights to the more important mutated models, thereby better utilizing mutation results for test prioritization. The superior performance of RFGP indicates that the random forest algorithm it utilizes can effectively identify important mutated models and assign them high weights.

The efficiency of GraphPrior (all the six approaches) is acceptable. Table 5.8 illustrates the efficiency of GraphPrior in comparison with other approaches. The time cost of GraphPrior can be decomposed into three phases, namely mutant generation, training, and execution. Mutant generation involves the production of mutated models based on retraining the original GNN model. The training time represents the average duration needed for training a ranking model. Finally, execution time denotes the average duration expended on test prioritization. By decomposing the time cost into these distinct phases, we provide a more detailed understanding of the efficiency of GraphPrior in contrast to other approaches. As evident from Table 5.8, the average execution time of GraphPrior for test prioritization is 40 seconds, with the most time-consuming phase being mutant generation, which takes around 35 minutes. In contrast, the average execution time of the compared approaches is less than one second. Although GraphPrior is not as efficient as the compared approaches, it provides a viable alternative to costly and time-consuming manual labeling, and its total time cost remains acceptable in real-world scenarios.

Answer to RQ2: *Among all the GraphPrior approaches, RFGP demonstrates the highest level of effectiveness in most cases. The efficiency of GraphPrior (all the six approaches) is acceptable.*

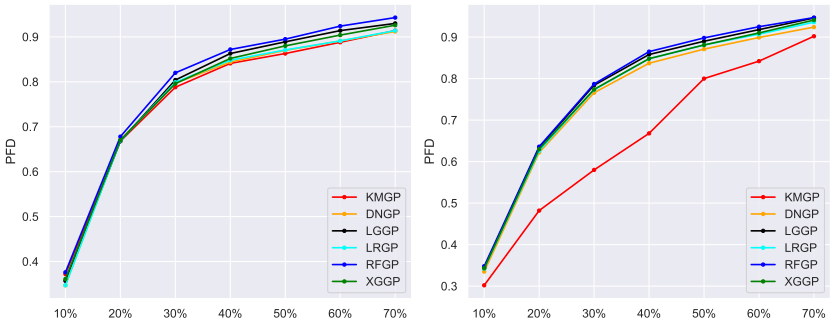
Table 5.5: Effectiveness comparison among KMGP and the feature-based GraphPrior approaches in terms of APFD

Data	Model	Approaches					
		DGGP	LGGP	XGGP	LRGP	RFGP	**KMGP**
CiteSeer	GAT	0.633	0.678	0.669	0.651	0.675	0.708
	GCN	0.682	0.695	0.690	0.678	0.694	0.701
	GraphSAGE	0.656	0.694	0.699	0.682	0.710	0.739
	TAGCN	0.652	0.681	0.694	0.660	0.696	0.712
Cora	GAT	0.749	0.785	0.795	0.767	0.811	0.841
	GCN	0.778	0.791	0.791	0.784	0.806	0.812
	GraphSAGE	0.764	0.791	0.793	0.784	0.794	0.792
	TAGCN	0.777	0.785	0.785	0.778	0.800	0.782
LastFM	GAT	0.799	0.814	0.812	0.802	0.826	0.801
	GCN	0.796	0.811	0.809	0.802	0.816	0.761
	GraphSAGE	0.771	0.785	0.780	0.778	0.789	0.702
	TAGCN	0.763	0.781	0.776	0.770	0.779	0.673
PubMed	GAT	0.740	0.774	0.768	0.763	0.773	0.735
	GCN	0.743	0.749	0.745	0.746	0.750	0.748
	GraphSAGE	0.743	0.776	0.767	0.768	0.774	0.747
	TAGCN	0.701	0.780	0.773	0.765	0.768	0.720
Average		0.734	0.761	0.759	0.749	0.766	0.748

Table 5.6: Effectiveness comparison among KMGP and the feature-based GraphPrior approaches in terms of PFD

Data	Model	Approaches	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60	PFD-70
CiteSeer	GAT	**KMGP**	0.264	0.464	0.629	0.750	0.812	0.841	0.875
		DNGP	0.252	0.460	0.596	0.647	0.693	0.722	0.753
		LGGP	0.251	0.465	0.621	0.715	0.759	0.791	0.833
		LRGP	0.244	0.467	0.611	0.683	0.721	0.750	0.788
		RFGP	0.257	0.470	0.619	0.697	0.743	0.781	0.827
		XGGP	0.256	0.464	0.618	0.702	0.740	0.771	0.817
	GCN	**KMGP**	0.278	0.492	0.652	0.723	0.771	0.811	0.865
		DNGP	0.248	0.479	0.643	0.699	0.748	0.794	0.843
		LGGP	0.273	0.483	0.651	0.717	0.764	0.803	0.856
		LRGP	0.251	0.484	0.643	0.698	0.745	0.787	0.832
		RFGP	0.272	0.486	0.653	0.716	0.762	0.807	0.852
		XGGP	0.265	0.481	0.650	0.711	0.760	0.804	0.848
	GraphSAGE	**KMGP**	0.306	0.525	0.679	0.774	0.835	0.879	0.910
		DNGP	0.271	0.511	0.635	0.670	0.695	0.729	0.771
		LGGP	0.287	0.515	0.680	0.733	0.767	0.797	0.831
		LRGP	0.273	0.512	0.671	0.708	0.737	0.767	0.806
		RFGP	0.287	0.515	0.684	0.730	0.775	0.816	0.865
		XGGP	0.283	0.516	0.661	0.703	0.753	0.800	0.851
	TAGCN	**KMGP**	0.295	0.490	0.617	0.723	0.795	0.845	0.888
		DNGP	0.285	0.504	0.578	0.628	0.682	0.740	0.784
		LGGP	0.298	0.513	0.651	0.700	0.737	0.773	0.811
		LRGP	0.292	0.507	0.587	0.640	0.692	0.749	0.799
		RFGP	0.294	0.511	0.662	0.694	0.747	0.793	0.845
		XGGP	0.297	0.510	0.636	0.695	0.748	0.801	0.849
Cora	GAT	**KMGP**	0.454	0.759	0.884	0.919	0.939	0.954	0.975
		DNGP	0.383	0.722	0.791	0.800	0.814	0.827	0.848
		LGGP	0.427	0.724	0.823	0.836	0.848	0.867	0.894
		LRGP	0.361	0.725	0.826	0.834	0.845	0.858	0.871
		RFGP	0.428	0.730	0.869	0.882	0.894	0.909	0.928
		XGGP	0.375	0.729	0.849	0.870	0.885	0.902	0.916
	GCN	**KMGP**	0.384	0.704	0.854	0.884	0.909	0.933	0.952
		DNGP	0.359	0.691	0.814	0.844	0.870	0.893	0.914
		LGGP	0.357	0.678	0.831	0.862	0.889	0.913	0.932
		LRGP	0.359	0.687	0.823	0.853	0.880	0.902	0.920
		RFGP	0.379	0.691	0.848	0.876	0.900	0.928	0.947
		XGGP	0.365	0.682	0.830	0.861	0.885	0.911	0.930
	GraphSAGE	**KMGP**	0.489	0.705	0.777	0.820	0.848	0.886	0.919
		DNGP	0.480	0.705	0.736	0.776	0.805	0.845	0.879
		LGGP	0.475	0.721	0.772	0.818	0.857	0.895	0.924
		LRGP	0.474	0.728	0.776	0.802	0.832	0.863	0.906
		RFGP	0.487	0.736	0.771	0.803	0.848	0.896	0.923
		XGGP	0.479	0.718	0.760	0.803	0.854	0.894	0.939
	TAGCN	**KMGP**	0.372	0.668	0.788	0.841	0.863	0.888	0.914
		DNGP	0.347	0.671	0.797	0.844	0.870	0.891	0.912
		LGGP	0.357	0.668	0.804	0.863	0.889	0.914	0.930
		LRGP	0.347	0.669	0.799	0.848	0.871	0.891	0.914
		RFGP	0.376	0.678	0.820	0.872	0.895	0.924	0.943
		XGGP	0.361	0.670	0.796	0.852	0.880	0.904	0.926

Data	Model	Approaches	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60	PFD-70
LastFM	GAT	**KMGP**	0.389	0.683	0.810	0.869	0.902	0.927	0.945
		DNGP	0.382	0.728	0.848	0.863	0.883	0.905	0.926
		LGGP	0.397	0.740	0.861	0.889	0.904	0.924	0.942
		LRGP	0.389	0.729	0.848	0.876	0.892	0.910	0.929
		RFGP	0.404	0.746	0.874	0.906	0.927	0.944	0.960
		XGGP	0.393	0.737	0.856	0.886	0.901	0.921	0.942
	GCN	**KMGP**	0.403	0.648	0.728	0.770	0.830	0.868	0.915
		DNGP	0.412	0.717	0.814	0.849	0.877	0.906	0.93
		LGGP	0.428	0.730	0.830	0.873	0.898	0.921	0.945
		LRGP	0.424	0.717	0.817	0.859	0.886	0.912	0.937
		RFGP	0.431	0.735	0.842	0.881	0.906	0.927	0.949
		XGGP	0.424	0.724	0.826	0.869	0.895	0.918	0.942
	GraphSAGE	**KMGP**	0.302	0.482	0.580	0.668	0.800	0.842	0.902
		DNGP	0.335	0.622	0.766	0.837	0.871	0.899	0.924
		LGGP	0.344	0.634	0.784	0.858	0.890	0.918	0.946
		LRGP	0.342	0.626	0.773	0.848	0.881	0.907	0.936
		RFGP	0.348	0.636	0.787	0.865	0.898	0.925	0.947
		XGGP	0.343	0.630	0.774	0.848	0.881	0.910	0.941
	TAGCN	**KMGP**	0.250	0.431	0.544	0.644	0.706	0.819	0.892
		DNGP	0.294	0.552	0.742	0.840	0.884	0.915	0.936
		LGGP	0.299	0.562	0.758	0.865	0.914	0.944	0.964
		LRGP	0.295	0.555	0.747	0.846	0.896	0.927	0.950
		RFGP	0.300	0.561	0.756	0.867	0.915	0.942	0.961
		XGGP	0.297	0.558	0.751	0.860	0.911	0.936	0.960
PubMed	GAT	**KMGP**	0.336	0.588	0.697	0.754	0.813	0.859	0.893
		DNGP	0.334	0.631	0.730	0.767	0.803	0.843	0.883
		LGGP	0.363	0.643	0.763	0.816	0.853	0.893	0.932
		LRGP	0.354	0.632	0.746	0.803	0.841	0.881	0.919
		RFGP	0.362	0.639	0.763	0.815	0.853	0.894	0.929
		XGGP	0.360	0.640	0.756	0.806	0.844	0.886	0.921
	GCN	**KMGP**	0.347	0.607	0.743	0.788	0.826	0.860	0.894
		DNGP	0.347	0.629	0.739	0.779	0.816	0.851	0.885
		LGGP	0.355	0.634	0.746	0.785	0.823	0.857	0.891
		LRGP	0.353	0.629	0.741	0.782	0.818	0.854	0.888
		RFGP	0.354	0.629	0.745	0.787	0.824	0.858	0.892
		XGGP	0.348	0.629	0.740	0.780	0.818	0.853	0.886
	GraphSAGE	**KMGP**	0.396	0.635	0.713	0.757	0.808	0.850	0.889
		DNGP	0.396	0.670	0.717	0.753	0.791	0.833	0.872
		LGGP	0.409	0.684	0.758	0.803	0.843	0.882	0.917
		LRGP	0.406	0.677	0.744	0.791	0.833	0.874	0.909
		RFGP	0.408	0.684	0.758	0.802	0.843	0.881	0.914
		XGGP	0.404	0.678	0.748	0.793	0.832	0.871	0.907
	TAGCN	**KMGP**	0.379	0.545	0.610	0.722	0.791	0.844	0.885
		DNGP	0.402	0.593	0.644	0.692	0.734	0.777	0.828
		LGGP	0.415	0.631	0.731	0.804	0.865	0.910	0.946
		LRGP	0.409	0.618	0.707	0.784	0.840	0.889	0.927
		RFGP	0.409	0.621	0.722	0.795	0.847	0.889	0.923
		XGGP	0.410	0.627	0.722	0.796	0.852	0.899	0.934



a) Cora, TAGCN **b)** LastFM, GraphSAGE

Figure 5.3: Test prioritization effectiveness of the six GraphPrior approaches for Cora with TAGCN and LastFM with GraphSAGE. X-Axis: the percentage of prioritized tests; Y-Axis: the percentage of detected miscalssified tests

Table 5.7: Average effectiveness comparison among KMGP and the feature-based GraphPrior approaches in terms of PFD

Approaches	Average PFD						
	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60	PFD-70
KMGP	0.353	0.589	0.707	0.775	0.828	0.869	0.907
DNGP	0.346	0.618	0.724	0.768	0.802	0.836	0.868
LGGP	0.358	0.627	0.754	0.809	0.844	0.875	0.906
LRGP	0.348	0.623	0.741	0.791	0.826	0.858	0.890
RFGP	0.362	0.629	0.761	0.812	0.849	0.882	0.913
XGGP	0.354	0.624	0.748	0.802	0.840	0.874	0.907

5.5.3 RQ3: Effectiveness of GraphPrior on adversarial test inputs

Objectives: We further investigate the effectiveness of GraphPrior on adversarial test data. Here, we adopt eight graph adversarial attacks (cf. Section 5.4.4) from the existing studies [173, 174, 175, 176]. The results can answer whether GraphPrior can perform well on adversarial test sets for GNNs, compared with existing approaches that can be used to identify possibly-misclassified test inputs.

Experimental design: We evaluate GraphPrior on adversarial datasets generated by 8 graph attack techniques [173] [174] [175] [176]. In this research question, we set the attack level as 0.3, which means that 30% of the test inputs in the test set are adversarial tests. It is important to note that a high attack level, such as 90%, would result in a significant ratio of adversarial test inputs. Under such circumstances, a larger number of bug cases could be selected by any of the prioritization methods, making it difficult to demonstrate the effectiveness of GraphPrior. Thus, in order to ensure an effective evaluation of GraphPrior and the compared approaches, we selected a reasonable attack level (i.e., 0.3), which can limit the proportion of adversarial test inputs. Totally, in this research question, we evaluate GraphPrior on 108 subjects (4 GNN models, 4 datasets and 8 graph adversarial attacks). We then ran all six GraphPrior approaches and the compared approaches on the subjects, and calculated the APFD values of each approach with each graph adversarial attack. Moreover, we calculated the PFD values of each approach in terms of different ratios of prioritized values.

Table 5.8: Time comparison between GraphPrior and compared approaches

Time cost parts	Approaches							
	GraphPrior	DeepGini	Least Confidence	Margin	Vanilla SM	PCS	Entropy	Rndom
Mutant Generation	**35 min**	-	-	-	-	-	-	-
Training	**3 min**	-	-	-	-	-	-	-
Execution	**40 s**	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s

Table 5.9: Effectiveness comparison among GraphPrior and the compared approaches in terms of APFD

Attack	Approaches												
	DNGP	**KMGP**	**LGGP**	**XGGP**	**LRGP**	**RFGP**	DeepGini	Least Confidence	Margin	Random	Vanilla SM	PCS	Entropy
DICE	0.672	0.710	0.707	0.706	0.695	0.713	0.667	0.698	0.693	0.500	0.698	0.693	0.642
MMA	0.691	0.725	0.721	0.724	0.705	0.731	0.684	0.717	0.718	0.499	0.717	0.718	0.672
NEAA	0.698	0.723	0.733	0.732	0.721	0.738	0.676	0.711	0.703	0.499	0.711	0.703	0.646
NEAR	0.737	0.735	0.767	0.764	0.757	0.774	0.678	0.719	0.717	0.499	0.719	0.717	0.644
PGD	0.718	0.730	0.743	0.743	0.729	0.753	0.693	0.728	0.727	0.498	0.728	0.727	0.656
RAA	0.659	0.701	0.697	0.696	0.684	0.703	0.671	0.702	0.695	0.499	0.702	0.695	0.648
RAF	0.657	0.702	0.696	0.696	0.683	0.703	0.670	0.701	0.694	0.500	0.701	0.694	0.646
RAR	0.703	0.724	0.735	0.734	0.723	0.742	0.673	0.708	0.707	0.498	0.708	0.707	0.645
Average	**0.692**	**0.718**	**0.725**	**0.724**	**0.712**	**0.732**	**0.677**	**0.711**	**0.707**	**0.499**	**0.711**	**0.707**	**0.650**

Results: GraphPrior approaches outperform the compared approaches (i.e., DeepGini, Least Confidence, Margin, Vanilla SM, PCS, Entropy and Random) in the context of graph adversarial attacks. Table 5.9 shows the test prioritization effectiveness (measured by APFD) of GraphPrior and the compared approaches across a variety of adversarial attacks. The experimental results indicate that the GraphPrior approaches exhibit superior performance, with the average APFD values ranging from 0.692 to 0.732, while the compared approaches range from 0.499 to 0.711. Notably, five GraphPrior approaches, namely RFGP, XGGP, LRGP, LGGP, and KMGP, outperform all the compared approaches on average across all the adversarial attacks. Table 5.10 presents the comparison results of GraphPrior and the compared approaches in terms of PFD, confirming the superior performance of GraphPrior from both the perspective of average effectiveness and the number of best cases. Furthermore, Table 5.11 presents the overall comparison results in terms of PFD, which further support the above conclusions by demonstrating that the largest average effectiveness of each case is achieved by the GraphPrior approaches, along with the largest number of best cases.

Among all the GraphPrior approaches proposed, the effectiveness of RFGP stands out as the most notable. From Table 5.9, in which the effectiveness is measured by the APFD values, we see that RFGP performs the best across different adversarial attacks, with the average improvement of 2.95%~46.69% compared with uncertainty-based test prioritization approaches. Table 5.10 presents the test prioritization effectiveness in terms of PFD. The column #Best case in PFD denotes the number of best cases a test prioritization approach achieved across all cases (i.e., all subjects of a graph adversarial attack). The results demonstrate that, against a majority of adversarial attacks, RFGP consistently outperforms all other GraphPrior approaches in terms of average effectiveness. Moreover, Table 5.11 presents the overall comparison results in terms of PFD, further indicating that RFGP outperforms all other approaches in terms of average effectiveness. Notably, when prioritizing 20% to 40% of the test inputs, RFGP consistently exhibits the highest number of best cases across a variety of subjects.

Answer to RQ3: *GraphPrior approaches outperform the compared approaches (i.e., DeepGini, Least Confidence, Margin, Vanilla SM, PCS, Entropy and Random) in the context of graph adversarial attacks. Among all the GraphPrior approaches proposed, the effectiveness of RFGP stands out as the most notable.*

5.5.4 RQ4: Effectiveness of GraphPrior against adversarial attacks at varying attack levels

Objectives: We investigate the effectiveness of GraphPrior on adversarial test inputs with different attack levels.

Table 5.10: Effectiveness comparison of GraphPrior and the compared approaches on adversarial test inputs in terms of PFD

Attack	Approaches	#Best cases in PFD				Average PFD				Attack	Approaches	#Best cases in PFD				Average PFD			
		PFD-10	PFD-20	PFD-30	PFD-40	PFD-10	PFD-20	PFD-30	PFD-40			PFD-10	PFD-20	PFD-30	PFD-40	PFD-10	PFD-20	PFD-30	PFD-40
	DNGP	0	2	0	0	0.289	0.553	0.705	0.769		**DNGP**	0	1	1	0	0.309	0.572	0.701	0.752
	KMGP	7	1	2	2	0.300	0.520	0.665	0.754		**KMGP**	7	4	3	3	0.336	0.573	0.714	0.789
	LGGP	1	0	0	0	0.301	0.557	0.719	0.801		**LGGP**	0	0	0	0	0.307	0.563	0.707	0.781
	LRGP	0	0	0	0	0.291	0.555	0.711	0.788		**LRGP**	0	2	0	0	0.305	0.570	0.697	0.762
	RFGP	4	9	10	10	0.304	0.561	0.729	0.818		**RFGP**	1	1	4	5	0.326	0.571	0.727	0.806
DICE	**XGGP**	0	0	0	0	0.293	0.556	0.716	0.799	PGD	**XGGP**	0	0	0	0	0.300	0.558	0.703	0.774
	DeepGini	0	0	0	0	0.215	0.394	0.535	0.655		DeepGini	0	0	0	0	0.238	0.413	0.556	0.672
	Entropy	0	0	0	0	0.212	0.381	0.507	0.611		Entropy	0	0	0	0	0.236	0.409	0.547	0.660
	Least Confidence	0	0	0	0	0.233	0.428	0.590	0.713		Least Confidence	0	0	0	0	0.255	0.451	0.604	0.727
	Margin	0	0	0	0	0.225	0.423	0.584	0.711		Margin	0	0	0	0	0.242	0.449	0.606	0.730
	PCS	0	0	0	0	0.225	0.423	0.584	0.711		PCS	0	0	0	0	0.242	0.449	0.606	0.730
	Vanilla SM	0	0	0	0	0.233	0.428	0.590	0.713		Vanilla SM	0	0	0	0	0.255	0.451	0.604	0.727
	Random	0	0	0	0	0.100	0.200	0.299	0.398		Random	0	0	0	0	0.098	0.199	0.299	0.397
	DNGP	0	2	0	0	0.320	0.598	0.729	0.785		**DNGP**	0	1	0	0	0.303	0.573	0.719	0.781
	KMGP	7	5	5	4	0.340	0.578	0.701	0.773		**KMGP**	5	3	2	4	0.308	0.544	0.675	0.755
	LGGP	1	1	0	0	0.327	0.597	0.739	0.809		**LGGP**	6	4	4	3	0.314	0.578	0.734	0.812
	LRGP	0	0	0	0	0.320	0.595	0.729	0.793		**LRGP**	0	1	0	0	0.307	0.574	0.727	0.800
	RFGP	4	4	7	8	0.341	0.598	0.754	0.829		**RFGP**	5	7	10	9	0.315	0.579	0.737	0.821
MMA	**XGGP**	0	0	0	0	0.319	0.592	0.736	0.804	RAA	**XGGP**	0	0	0	0	0.307	0.574	0.730	0.808
	DeepGini	0	0	0	0	0.243	0.426	0.568	0.682		DeepGini	0	0	0	0	0.221	0.395	0.538	0.652
	Entropy	0	0	0	0	0.240	0.412	0.538	0.635		Entropy	0	0	0	0	0.219	0.387	0.518	0.623
	Least Confidence	0	0	0	0	0.263	0.469	0.622	0.741		Least Confidence	0	0	0	0	0.234	0.425	0.582	0.705
	Margin	0	0	0	0	0.253	0.463	0.622	0.743		Margin	0	0	0	0	0.220	0.411	0.570	0.698
	PCS	0	0	0	0	0.253	0.463	0.622	0.743		PCS	0	0	0	0	0.220	0.411	0.570	0.698
	Vanilla SM	0	0	0	0	0.263	0.469	0.622	0.741		Vanilla SM	0	0	0	0	0.234	0.425	0.582	0.705
	Random	0	0	0	0	0.102	0.202	0.303	0.402		Random	0	0	0	0	0.101	0.201	0.301	0.399
	DNGP	0	0	0	0	0.332	0.627	0.783	0.840		**DNGP**	0	1	0	0	0.295	0.565	0.715	0.780
	KMGP	3	2	2	1	0.335	0.589	0.733	0.805		**KMGP**	7	3	1	3	0.301	0.533	0.673	0.760
	LGGP	0	2	0	1	0.343	0.636	0.803	0.877		**LGGP**	4	5	5	4	0.307	0.568	0.731	0.812
	LRGP	1	0	0	0	0.334	0.630	0.795	0.860		**LRGP**	0	0	0	0	0.298	0.565	0.723	0.798
	RFGP	4	4	6	6	0.345	0.640	0.814	0.884		**RFGP**	5	7	10	9	0.308	0.570	0.736	0.821
NEAA	**XGGP**	0	0	0	0	0.336	0.631	0.800	0.869	RAF	**XGGP**	0	0	0	0	0.299	0.565	0.727	0.807
	DeepGini	0	0	0	0	0.245	0.433	0.579	0.694		DeepGini	0	0	0	0	0.218	0.394	0.536	0.650
	Entropy	0	0	0	0	0.240	0.414	0.538	0.632		Entropy	0	0	0	0	0.216	0.385	0.516	0.620
	Least Confidence	0	0	0	0	0.261	0.472	0.638	0.763		Least Confidence	0	0	0	0	0.230	0.422	0.580	0.706
	Margin	0	0	0	0	0.245	0.457	0.625	0.757		Margin	0	0	0	0	0.217	0.409	0.567	0.698
	PCS	0	0	0	0	0.245	0.457	0.625	0.757		PCS	0	0	0	0	0.217	0.409	0.567	0.698
	Vanilla SM	0	0	0	0	0.261	0.472	0.638	0.763		Vanilla SM	0	0	0	0	0.230	0.422	0.580	0.706
	Random	0	0	0	0	0.100	0.200	0.301	0.399		Random	0	0	0	0	0.100	0.202	0.301	0.402
	DNGP	0	0	0	0	0.322	0.618	0.787	0.848		**DNGP**	0	2	0	0	0.334	0.606	0.720	0.766
	KMGP	1	2	2	1	0.335	0.618	0.780	0.856		**KMGP**	6	1	1	4	0.341	0.568	0.697	0.772
	LGGP	1	0	0	0	0.336	0.620	0.793	0.871		**LGGP**	2	4	4	4	0.347	0.616	0.752	0.814
	LRGP	0	0	0	0	0.326	0.621	0.798	0.866		**LRGP**	1	0	0	0	0.338	0.611	0.740	0.799
	RFGP	2	2	2	3	0.339	0.627	0.810	0.893		**RFGP**	7	8	11	7	0.348	0.617	0.761	0.823
NEAR	**XGGP**	0	0	0	0	0.327	0.620	0.796	0.872	RAR	**XGGP**	0	1	0	1	0.339	0.613	0.749	0.810
	DeepGini	0	0	0	0	0.247	0.432	0.576	0.687		DeepGini	0	0	0	0	0.231	0.410	0.551	0.662
	Entropy	0	0	0	0	0.244	0.427	0.569	0.675		Entropy	0	0	0	0	0.229	0.400	0.528	0.627
	Least Confidence	0	0	0	0	0.256	0.458	0.621	0.747		Least Confidence	0	0	0	0	0.247	0.445	0.603	0.723
	Margin	0	0	0	0	0.233	0.431	0.600	0.737		Margin	0	0	0	0	0.243	0.444	0.605	0.727
	PCS	0	0	0	0	0.233	0.431	0.600	0.737		PCS	0	0	0	0	0.243	0.444	0.605	0.727
	Vanilla SM	0	0	0	0	0.256	0.458	0.621	0.747		Vanilla SM	0	0	0	0	0.247	0.445	0.603	0.723
	Random	0	0	0	0	0.101	0.198	0.294	0.391		Random	0	0	0	0	0.099	0.200	0.300	0.401

Table 5.11: Average effectiveness comparision among GraphPrior and the compared approaches on adversarial test inputs in terms of PFD

Approaches	#Best case in PFD				Average PFD			
	PFD-10	PFD-20	PFD-30	PFD-40	PFD-10	PFD-20	PFD-30	PFD-40
DNGP	0	9	1	0	0.313	0.589	0.732	0.790
KMGP	43	21	18	22	0.325	0.565	0.705	0.783
LGGP	15	16	13	12	0.323	0.592	0.747	0.822
LRGP	2	3	0	0	0.315	0.590	0.740	0.808
RFGP	32	42	60	57	0.328	0.595	0.758	0.837
XGGP	0	1	0	1	0.315	0.589	0.745	0.818
DeepGini	0	0	0	0	0.232	0.412	0.555	0.669
Entropy	0	0	0	0	0.23	0.402	0.533	0.635
Least Confidence	0	0	0	0	0.247	0.446	0.605	0.728
Margin	0	0	0	0	0.235	0.436	0.597	0.725
PCS	0	0	0	0	0.235	0.436	0.597	0.725
Vanilla SM	0	0	0	0	0.247	0.446	0.605	0.728
Random	0	0	0	0	0.101	0.202	0.301	0.399

Experimental design: To investigate the effectiveness of GraphPrior on test inputs generated via different levels of graph adversarial attacks, we set different attack levels (i.e., 0.1, 0.2, 0.3 and 0.4) on 8 graph adversarial techniques (i.e., DICE, Min-max attack, NEAA, NEAR, PGD attack, RAA, RAF, and RAR). As mentioned in RQ3, the attack level indicates the ratio of adversarial inputs in the dataset. For example, 0.4 means that 40% tests in the dataset are adversarial tests. We select these attack levels because a high attack level (e.g., 80%) would engender a substantial proportion of adversarial test inputs. Consequently, such circumstances could yield a greater number of bug cases selected by any prioritization method, thereby affecting the evaluation of GraphPrior. Therefore, we carefully selected a range of attack levels that are not unduly high for the evaluation of GraphPrior. In this research question, we totally evaluate GraphPrior and the compared approaches on 432 subjects.

Results: GraphPrior outperforms all the compared approaches on the adversarial test inputs generated from different attack levels. More specifically, Table 5.12 presents the effectiveness of GraphPrior and the compared approaches under the attacks DICE, MMA, RAA and RAR, with the attack level ranging from 0.1 to 0.4. In this research question, we totally apply 8 adversarial attacks. The remaining experimental results (i.e., results of the other four adversarial attacks) are presented on our Github[2].

The experimental results presented in Table 5.12 demonstrate that GraphPrior, consisting of DNGP, KMGP, LGGP, LRGP, RFGP and XGGP, outperforms all the compared approaches across different levels of the adversarial attacks.

Table 5.13 demonstrates the overall comparison results among GraphPrior and the compared approaches across 8 adversarial attacks with differnet attack levels. Specifically, we evaluate the effectiveness of each test prioritization approach in terms of the number of cases where it performed the best, as well as its average PFD values across different attack levels. For example, the "All-0.1" refers to the overall results of each approach under all the adversarial attacks with an attack level of 0.1. Table 5.13 demonstrates that GraphPrior outperforms all compared approaches, achieving the best effectiveness in 99.94% of the tested cases. Only one best case is achieved by the compared approach margin. Furthermore, GraphPrior approaches such as RFGP and KMGP consistently exhibit the largest average PFD values across different attack levels.

Among all the GraphPrior approaches, RFGP and KMGP exhibit superior performance across different attack levels in comparison to other GraphPrior approaches. In Table 5.12, we see that, across the attack levels from 0.1 to 0.4, RFGP performs the best in the largest number of best cases, followed by KMGP. For example, when the attack level is 0.1, RFGP performs the best in 46.47% cases. KMGP performs the best in 35.33% cases. Notably, when prioritizing 10% test inputs, KMGP takes the largest number of best cases. When the attack level is 0.2~0.4, RFGP takes the largest number of best cases.

Additionally, our experimental results, as illustrated in Table 5.13, reveal that the RFGP technique exhibits the largest average PFD values when compared to the other evaluated approaches across varying attack levels. Specifically, when 40% of the test inputs are prioritized, RFGP achieves a PFD value ranging from 0.832 to 0.836, which indicates the ability to detect more than 80% of misclassified tests.

[2]https://github.com/yinghuali/GraphPrior/tree/main/mutation/adv_res

Answer to RQ4: *GraphPrior outperforms all the compared approaches on the adversarial test inputs generated from different attack levels. Among all the GraphPrior approaches, RFGP and KMGP exhibit superior performance across different attack levels in comparison to other GraphPrior approaches.*

5.5.5 RQ5: Contribution analysis of different mutation rules

Objectives: For each evaluated GNN model, we investigate which mutated rules generate more top contributing mutated models for test prioritization.

Experimental design: In our study, we employed one or more mutation rules to generate a mutated model. Each mutated model corresponds to one mutation feature. Thus, to evaluate the importance of different mutation rules, we initially evaluate the importance of various mutation features. We adopted the cover metric of the XGBoost algorithm to identify the importance of each mutation feature for ranking models. A detailed account of this approach is presented in Section 4.5. After computing the importance scores of all the mutated features, we selected the top-N important features for each subject and subsequently identified the top-N mutated models. We then identified the mutation rules utilized to generate each mutated model and compared the contributions of the mutation rules accordingly. Additionally, for different subjects in this research question, we generate 80~240 mutated models.

Results: The mutation rule HC made high contributions to the effectiveness of GraphPrior on all the four types of GNN models. Table 5.14 to Table 5.17 illustrate the contributions of different mutation rules to the effectiveness of GraphPrior on different GNN models (i.e., GCN, GAT, GraphSAGE and TAGCN). For each GNN model, we identify the top-N mutated models that made top contributions to the effectiveness of GraphPrior. The corresponding mutation rules applied to generate each mutated model are highlighted in grey. Table 5.14 presents the contributions of Top-N mutated models to the effectiveness of GraphPrior for the case of GCN model. Notably, the mutation rules **BIA** and **HC** made contributions to 100% of the top contributing mutated models, while **SL**, **NOR**, **CA**, and **IMP** contributed to a lower percentage of the top contributing mutated models. We conclude that, for the GCN model, the mutation rules **SL** and **HC** were the most effective in generating the top important mutated models. Moving to GAT, GraphSAGE, and TAGCN, whose results are presented in Table 5.15, Table 5.16, and Table 5.17, the mutation rule HC also generates a large ratio (i.e., 100%, 90%, and 90% respectively) of top contributing mutated models. We can conclude that, across the four different types of GNN models, HC can continuously make top contributions to the effectiveness of GraphPrior.

Some mutated rules, such as NOR and BIA, made high contributions to the effectiveness of GraphPrior on some specific GNN models. Moreover, some mutation rules, such as BIA and NOR, also generate a considerable ratio (i.e., from 50% to 100%) of top-critical mutated models. For example, on GCN and GraphSAGE, BIA made contributions to 100% top-N mutated models. On TAGCN, NOR made contributions to 100% top-N mutated models.

Table 5.12: Comparison results of GraphPrior and the compared approaches against different levels of the attacks DICE, MMA, RAA and RAR in terms of PFD

Attack	Approaches	#Best cases in PFD				Average PFD				Attack	Approaches	#Best cases in PFD				Average PFD			
		PFD-10	PFD-20	PFD-30	PFD-40	PFD-10	PFD-20	PFD-30	PFD-40			PFD-10	PFD-20	PFD-30	PFD-40	PFD-10	PFD-20	PFD-30	PFD-40
	DNGP	0	1	0	0	0.322	0.595	0.724	0.775		**DNGP**	0	0	0	0	0.333	0.604	0.725	0.774
	KMGP	7	3	2	5	0.335	0.568	0.700	0.776		**KMGP**	4	3	5	5	0.336	0.574	0.696	0.770
	LGGP	1	0	1	0	0.335	0.600	0.745	0.811		**LGGP**	3	6	4	4	0.343	0.611	0.751	0.814
	LRGP	0	0	0	0	0.323	0.597	0.736	0.797		**LRGP**	1	0	0	0	0.337	0.607	0.743	0.803
	RFGP	4	7	9	7	0.338	0.605	0.757	0.828		**RFGP**	8	7	7	7	0.345	0.613	0.757	0.822
DICE-0.1	**XGGP**	0	1	0	0	0.325	0.599	0.743	0.810	RAA-0.1	**XGGP**	0	0	0	0	0.338	0.608	0.749	0.813
	DeepGini	0	0	0	0	0.237	0.419	0.559	0.674		DeepGini	0	0	0	0	0.232	0.410	0.549	0.660
	Entropy	0	0	0	0	0.233	0.405	0.528	0.627		Entropy	0	0	0	0	0.230	0.399	0.527	0.626
	Least Confidence	0	0	0	0	0.256	0.459	0.616	0.736		Least Confidence	0	0	0	0	0.248	0.445	0.602	0.722
	Margin	0	0	0	0	0.245	0.451	0.613	0.737		Margin	0	0	0	0	0.236	0.438	0.597	0.720
	PCS	0	0	0	0	0.245	0.451	0.613	0.737		PCS	0	0	0	0	0.236	0.438	0.597	0.720
	Vanilla SM	0	0	0	0	0.256	0.459	0.616	0.736		Vanilla SM	0	0	0	0	0.248	0.445	0.602	0.722
	Random	0	0	0	0	0.098	0.198	0.296	0.397		Random	0	0	0	0	0.100	0.200	0.301	0.401
	DNGP	0	0	0	0	0.305	0.573	0.713	0.772		**DNGP**	0	0	0	0	0.311	0.584	0.717	0.773
	KMGP	6	3	2	3	0.314	0.545	0.678	0.762		**KMGP**	6	5	4	5	0.318	0.553	0.683	0.762
	LGGP	1	2	0	1	0.314	0.576	0.732	0.807		**LGGP**	4	5	3	3	0.323	0.587	0.736	0.807
	LRGP	0	0	0	0	0.304	0.575	0.724	0.795		**LRGP**	1	1	0	0	0.314	0.586	0.729	0.796
	RFGP	5	6	10	8	0.316	0.579	0.741	0.820		**RFGP**	5	5	9	8	0.324	0.590	0.744	0.818
DICE-0.2	**XGGP**	0	1	0	0	0.305	0.574	0.729	0.804	RAA-0.2	**XGGP**	0	0	0	0	0.314	0.586	0.734	0.805
	DeepGini	0	0	0	0	0.228	0.409	0.552	0.667		DeepGini	0	0	0	0	0.223	0.397	0.540	0.653
	Entropy	0	0	0	0	0.225	0.395	0.522	0.624		Entropy	0	0	0	0	0.221	0.388	0.519	0.620
	Least Confidence	0	0	0	0	0.244	0.443	0.602	0.724		Least Confidence	0	0	0	0	0.237	0.431	0.588	0.713
	Margin	0	0	0	0	0.235	0.435	0.596	0.723		Margin	0	0	0	0	0.226	0.422	0.582	0.709
	PCS	0	0	0	0	0.235	0.435	0.596	0.723		PCS	0	0	0	0	0.226	0.422	0.582	0.709
	Vanilla SM	0	0	0	0	0.244	0.443	0.602	0.724		Vanilla SM	0	0	0	0	0.237	0.431	0.588	0.713
	Random	0	0	0	0	0.101	0.202	0.302	0.401		Random	0	0	0	0	0.099	0.199	0.298	0.398
	DNGP	0	2	0	0	0.289	0.553	0.705	0.769		**DNGP**	0	1	0	0	0.303	0.573	0.719	0.781
	KMGP	7	1	2	2	0.300	0.520	0.665	0.754		**KMGP**	5	3	2	4	0.308	0.544	0.675	0.755
	LGGP	1	0	0	0	0.301	0.557	0.719	0.801		**LGGP**	6	4	4	3	0.314	0.578	0.734	0.812
	LRGP	0	0	0	0	0.291	0.555	0.711	0.788		**LRGP**	0	1	0	0	0.307	0.574	0.727	0.800
	RFGP	4	9	10	10	0.304	0.561	0.729	0.818		**RFGP**	5	7	10	9	0.315	0.579	0.737	0.821
DICE-0.3	**XGGP**	0	0	0	0	0.293	0.556	0.716	0.799	RAA-0.3	**XGGP**	0	0	0	0	0.307	0.574	0.730	0.808
	DeepGini	0	0	0	0	0.215	0.394	0.535	0.655		DeepGini	0	0	0	0	0.221	0.395	0.538	0.652
	Entropy	0	0	0	0	0.212	0.381	0.507	0.611		Entropy	0	0	0	0	0.219	0.387	0.518	0.623
	Least Confidence	0	0	0	0	0.233	0.428	0.590	0.713		Least Confidence	0	0	0	0	0.234	0.425	0.582	0.705
	Margin	0	0	0	0	0.225	0.423	0.584	0.711		Margin	0	0	0	0	0.220	0.411	0.570	0.698
	PCS	0	0	0	0	0.225	0.423	0.584	0.711		PCS	0	0	0	0	0.220	0.411	0.570	0.698
	Vanilla SM	0	0	0	0	0.233	0.428	0.590	0.713		Vanilla SM	0	0	0	0	0.234	0.425	0.582	0.705
	Random	0	0	0	0	0.100	0.200	0.299	0.398		Random	0	0	0	0	0.101	0.201	0.301	0.399
	DNGP	0	1	2	0	0.276	0.532	0.694	0.770		**DNGP**	0	1	0	0	0.290	0.554	0.713	0.783
	KMGP	7	2	1	1	0.288	0.510	0.647	0.740		**KMGP**	6	3	1	4	0.294	0.525	0.671	0.761
	LGGP	0	1	1	1	0.286	0.535	0.702	0.799		**LGGP**	4	5	3	3	0.300	0.559	0.726	0.812
	LRGP	0	0	1	0	0.277	0.533	0.699	0.785		**LRGP**	0	1	0	0	0.293	0.556	0.720	0.800
	RFGP	5	8	7	10	0.291	0.538	0.708	0.812		**RFGP**	6	6	12	9	0.302	0.560	0.731	0.823
DICE-0.4	**XGGP**	0	0	0	0	0.280	0.532	0.700	0.795	RAA-0.4	**XGGP**	0	0	0	0	0.294	0.556	0.724	0.809
	DeepGini	0	0	0	0	0.211	0.388	0.533	0.654		DeepGini	0	0	0	0	0.215	0.392	0.535	0.650
	Entropy	0	0	0	0	0.209	0.376	0.508	0.613		Entropy	0	0	0	0	0.213	0.384	0.517	0.621
	Least Confidence	0	0	0	0	0.226	0.419	0.579	0.708		Least Confidence	0	0	0	0	0.226	0.418	0.576	0.702
	Margin	0	0	0	0	0.215	0.406	0.568	0.701		Margin	0	0	0	0	0.210	0.399	0.559	0.689
	PCS	0	0	0	0	0.215	0.406	0.568	0.701		PCS	0	0	0	0	0.210	0.399	0.559	0.689
	Vanilla SM	0	0	0	0	0.226	0.419	0.579	0.708		Vanilla SM	0	0	0	0	0.226	0.418	0.576	0.702
	Random	0	0	0	0	0.098	0.200	0.300	0.400		Random	0	0	0	0	0.098	0.200	0.300	0.400
	DNGP	0	1	0	0	0.329	0.611	0.733	0.781		**DNGP**	0	0	0	0	0.342	0.613	0.721	0.767
	KMGP	7	5	4	4	0.346	0.583	0.708	0.776		**KMGP**	6	3	3	7	0.348	0.583	0.704	0.774
	LGGP	0	1	2	1	0.336	0.609	0.748	0.810		**LGGP**	4	4	4	1	0.352	0.621	0.753	0.809
	LRGP	0	0	0	0	0.330	0.608	0.738	0.800		**LRGP**	1	0	0	0	0.342	0.616	0.740	0.792
	RFGP	5	5	6	7	0.349	0.614	0.763	0.833		**RFGP**	5	9	8	8	0.357	0.624	0.760	0.817
MMA-0.1	**XGGP**	0	0	0	0	0.329	0.607	0.746	0.808	RAR-0.1	**XGGP**	0	0	1	0	0.345	0.620	0.748	0.806
	DeepGini	0	0	0	0	0.246	0.428	0.568	0.682		DeepGini	0	0	0	0	0.240	0.416	0.552	0.662
	Entropy	0	0	0	0	0.241	0.413	0.536	0.634		Entropy	0	0	0	0	0.238	0.404	0.527	0.626
	Least Confidence	0	0	0	0	0.269	0.472	0.627	0.745		Least Confidence	0	0	0	0	0.257	0.455	0.609	0.726
	Margin	0	0	0	0	0.257	0.467	0.627	0.749		Margin	0	0	0	0	0.248	0.451	0.609	0.729
	PCS	0	0	0	0	0.257	0.467	0.627	0.749		PCS	0	0	0	0	0.249	0.451	0.609	0.729
	Vanilla SM	0	0	0	0	0.269	0.472	0.627	0.745		Vanilla SM	0	0	0	0	0.257	0.455	0.609	0.726
	Random	0	0	0	0	0.099	0.198	0.296	0.396		Random	0	0	0	0	0.100	0.200	0.301	0.401
	DNGP	0	0	0	0	0.328	0.605	0.725	0.775		**DNGP**	0	1	0	0	0.341	0.614	0.723	0.771
	KMGP	6	5	5	4	0.344	0.581	0.705	0.774		**KMGP**	7	3	0	6	0.346	0.579	0.701	0.775
	LGGP	1	2	1	0	0.336	0.605	0.741	0.805		**LGGP**	4	3	3	3	0.353	0.619	0.751	0.812
	LRGP	0	0	0	0	0.331	0.603	0.734	0.794		**LRGP**	1	1	0	0	0.342	0.615	0.740	0.795
	RFGP	5	5	6	7	0.349	0.610	0.758	0.829		**RFGP**	4	7	13	7	0.356	0.623	0.762	0.822
MMA-0.2	**XGGP**	0	0	0	0	0.330	0.601	0.738	0.802	RAR-0.2	**XGGP**	0	1	0	0	0.343	0.618	0.748	0.805
	DeepGini	0	0	0	0	0.248	0.431	0.573	0.686		DeepGini	0	0	0	0	0.237	0.411	0.550	0.659
	Entropy	0	0	0	0	0.245	0.417	0.541	0.639		Entropy	0	0	0	0	0.234	0.401	0.526	0.624
	Least Confidence	0	0	0	0	0.267	0.473	0.629	0.746		Least Confidence	0	0	0	0	0.250	0.449	0.604	0.725
	Margin	0	0	0	1	0.255	0.466	0.626	0.746		Margin	0	0	0	0	0.244	0.447	0.605	0.728
	PCS	0	0	0	0	0.255	0.466	0.626	0.746		PCS	0	0	0	0	0.244	0.447	0.605	0.728
	Vanilla SM	0	0	0	0	0.267	0.473	0.629	0.746		Vanilla SM	0	0	0	0	0.250	0.449	0.604	0.725
	Random	0	0	0	0	0.099	0.199	0.297	0.401		Random	0	0	0	0	0.098	0.199	0.297	0.397
	DNGP	0	2	0	0	0.320	0.598	0.729	0.785		**DNGP**	0	2	0	0	0.334	0.606	0.720	0.766
	KMGP	7	5	5	4	0.340	0.578	0.701	0.773		**KMGP**	6	1	1	4	0.341	0.568	0.697	0.772
	LGGP	1	1	0	0	0.327	0.597	0.739	0.809		**LGGP**	2	4	4	4	0.347	0.616	0.752	0.814
	LRGP	0	0	0	0	0.320	0.595	0.729	0.793		**LRGP**	1	0	0	0	0.338	0.611	0.740	0.799
	RFGP	4	4	7	8	0.341	0.598	0.754	0.829		**RFGP**	7	8	11	7	0.348	0.617	0.761	0.823
MMA-0.3	**XGGP**	0	0	0	0	0.319	0.592	0.736	0.804	RAR-0.3	**XGGP**	0	1	0	1	0.339	0.613	0.749	0.810
	DeepGini	0	0	0	0	0.243	0.426	0.568	0.682		DeepGini	0	0	0	0	0.231	0.410	0.551	0.662
	Entropy	0	0	0	0	0.240	0.412	0.538	0.635		Entropy	0	0	0	0	0.229	0.400	0.528	0.627
	Least Confidence	0	0	0	0	0.263	0.469	0.622	0.741		Least Confidence	0	0	0	0	0.247	0.445	0.603	0.723
	Margin	0	0	0	0	0.253	0.463	0.622	0.743		Margin	0	0	0	0	0.243	0.444	0.605	0.727
	PCS	0	0	0	0	0.253	0.463	0.622	0.743		PCS	0	0	0	0	0.243	0.444	0.605	0.727
	Vanilla SM	0	0	0	0	0.263	0.469	0.622	0.741		Vanilla SM	0	0	0	0	0.247	0.445	0.603	0.723
	Random	0	0	0	0	0.102	0.202	0.303	0.402		Random	0	0	0	0	0.099	0.200	0.300	0.401
	DNGP	0	2	0	0	0.322	0.601	0.732	0.787		**DNGP**	0	1	0	0	0.333	0.607	0.723	0.770
	KMGP	7	6	5	5	0.345	0.582	0.711	0.778		**KMGP**	7	1	1	3	0.337	0.564	0.692	0.771
	LGGP	1	3	2	0	0.331	0.598	0.739	0.807		**LGGP**	3	5	4	3	0.341	0.611	0.749	0.815
	LRGP	0	0	0	0	0.324	0.597	0.728	0.790		**LRGP**	0	2	0	0	0.335	0.609	0.738	0.799
	RFGP	4	1	5	7	0.345	0.598	0.754	0.827		**RFGP**	6	7	10	10	0.345	0.613	0.758	0.824
MMA-0.4	**XGGP**	0	0	0	0	0.323	0.592	0.731	0.797	RAR-0.4	**XGGP**	0	0	1	0	0.335	0.609	0.745	0.809
	DeepGini	0	0	0	0	0.245	0.427	0.568	0.681		DeepGini	0	0	0	0	0.233	0.406	0.547	0.658
	Entropy	0	0	0	0	0.242	0.413	0.539	0.636		Entropy	0	0	0	0	0.231	0.398	0.524	0.623
	Least Confidence	0	0	0	0	0.265	0.469	0.624	0.741		Least Confidence	0	0	0	0	0.248	0.439	0.596	0.719
	Margin	0	0	0	0	0.253	0.464	0.625	0.744		Margin	0	0	0	0	0.243	0.443	0.600	0.723
	PCS	0	0	0	0	0.253	0.464	0.625	0.744		PCS	0	0	0	0	0.243	0.443	0.600	0.723
	Vanilla SM	0	0	0	0	0.265	0.469	0.624	0.741		Vanilla SM	0	0	0	0	0.248	0.439	0.596	0.719
	Random	0	0	0	0	0.098	0.201	0.300	0.399		Random	0	0	0	0	0.097	0.199	0.299	0.398

Table 5.13: Overall comparison results among GraphPrior and the compared approaches on adversarial tests with different attack levels

Attack Level	Approaches	#Best case in PFD				Average PFD			
		PFD-10	PFD-20	PFD-30	PFD-40	PFD-10	PFD-20	PFD-30	PFD-40
All-0.1	**DNGP**	0	3	0	0	0.334	0.615	0.738	0.784
	KMGP	42	27	28	33	0.349	0.594	0.723	0.791
	LGGP	13	18	14	11	0.346	0.619	0.760	0.820
	LRGP	3	0	0	0	0.336	0.617	0.752	0.808
	RFGP	34	43	48	46	0.352	0.624	0.772	0.836
	XGGP	0	1	2	1	0.336	0.617	0.758	0.819
	DeepGini	0	0	0	0	0.243	0.425	0.566	0.679
	Entropy	0	0	0	0	0.241	0.413	0.541	0.642
	Least Confidence	0	0	0	0	0.261	0.465	0.623	0.742
	Margin	0	0	0	1	0.249	0.457	0.619	0.742
	PCS	0	0	0	0	0.249	0.457	0.619	0.742
	Vanilla SM	0	0	0	0	0.261	0.465	0.623	0.742
	Random	0	0	0	0	0.099	0.200	0.301	0.402
All-0.2	**DNGP**	0	2	0	0	0.323	0.602	0.734	0.786
	KMGP	44	29	20	29	0.335	0.580	0.713	0.786
	LGGP	13	19	10	10	0.332	0.604	0.753	0.820
	LRGP	2	3	0	0	0.323	0.602	0.745	0.806
	RFGP	33	37	62	52	0.339	0.608	0.765	0.836
	XGGP	0	2	0	0	0.323	0.602	0.750	0.816
	DeepGini	0	0	0	0	0.238	0.419	0.561	0.675
	Entropy	0	0	0	0	0.235	0.408	0.538	0.640
	Least Confidence	0	0	0	0	0.254	0.456	0.614	0.736
	Margin	0	0	0	1	0.241	0.446	0.609	0.734
	PCS	0	0	0	0	0.241	0.446	0.609	0.734
	Vanilla SM	0	0	0	0	0.254	0.456	0.614	0.736
	Random	0	0	0	0	0.099	0.199	0.299	0.399
All-0.3	**DNGP**	0	9	1	0	0.313	0.589	0.732	0.790
	KMGP	43	21	18	22	0.324	0.565	0.704	0.783
	LGGP	15	16	13	12	0.322	0.591	0.747	0.822
	LRGP	2	3	0	0	0.314	0.590	0.740	0.808
	RFGP	32	42	60	57	0.328	0.595	0.758	0.836
	XGGP	0	1	0	1	0.315	0.588	0.744	0.817
	DeepGini	0	0	0	0	0.232	0.412	0.554	0.669
	Entropy	0	0	0	0	0.229	0.401	0.532	0.635
	Least Confidence	0	0	0	0	0.247	0.446	0.605	0.728
	Margin	0	0	0	0	0.234	0.435	0.597	0.725
	PCS	0	0	0	0	0.234	0.435	0.597	0.725
	Vanilla SM	0	0	0	0	0.247	0.446	0.605	0.728
	Random	0	0	0	0	0.100	0.200	0.299	0.398
All-0.4	**DNGP**	0	8	3	0	0.306	0.578	0.727	0.790
	KMGP	43	23	15	23	0.316	0.554	0.694	0.776
	LGGP	13	20	13	11	0.314	0.580	0.739	0.819
	LRGP	2	3	1	0	0.307	0.577	0.732	0.805
	RFGP	34	38	58	58	0.320	0.581	0.748	0.832
	XGGP	0	0	2	0	0.307	0.576	0.735	0.813
	DeepGini	0	0	0	0	0.228	0.408	0.552	0.669
	Entropy	0	0	0	0	0.226	0.399	0.532	0.636
	Least Confidence	0	0	0	0	0.242	0.439	0.599	0.724
	Margin	0	0	0	0	0.227	0.426	0.588	0.717
	PCS	0	0	0	0	0.227	0.426	0.588	0.717
	Vanilla SM	0	0	0	0	0.242	0.439	0.599	0.724
	Random	0	0	0	0	0.097	0.199	0.299	0.398

Answer to RQ5: *The mutation rule HC made high contributions to the effectiveness of GraphPrior on all the four types of GNN models. Some mutated rules, such as NOR and BIA made high contributions to the effectiveness of GraphPrior on some specific GNN models.*

Table 5.14: The contributions of different mutation rules (GCN)

Top-N	SL	BIA	CA	IMP	NOR	HC
0	✓	✓				✓
1	✓	✓	✓	✓		✓
2		✓	✓	✓		✓
3		✓				✓
4	✓	✓		✓	✓	✓
5	✓	✓	✓	✓	✓	✓
6	✓	✓			✓	✓
7	✓	✓	✓		✓	✓
8	✓	✓	✓			✓
9	✓	✓			✓	✓

Table 5.15: The contributions of different mutation rules (GAT)

Top-N	SL	BIA	CON	HDS	EP	NS	HC
0	✓	✓	✓	✓	✓	✓	✓
1	✓		✓	✓	✓		✓
2			✓		✓		✓
3	✓	✓	✓	✓		✓	✓
4	✓	✓	✓		✓	✓	✓
5		✓	✓		✓		✓
6			✓		✓	✓	✓
7			✓	✓		✓	✓
8		✓	✓		✓	✓	✓
9		✓	✓	✓	✓		✓

Table 5.16: The contributions of different mutation rules (GraphSAGE)

Top-N	BIA	NOR	HC	EP
0	✓	✓	✓	✓
1	✓	✓	✓	
2	✓	✓	✓	✓
3	✓	✓	✓	
4	✓	✓	✓	
5	✓	✓	✓	
6	✓		✓	✓
7	✓	✓	✓	
8	✓	✓	✓	✓
9	✓	✓	✓	

Table 5.17: The contributions of different mutation rules to the (TAGCN)

Top-N	NOR	HC	EP
0	✓		
1	✓	✓	✓
2	✓	✓	✓
3	✓	✓	
4	✓	✓	✓
5	✓	✓	
6	✓	✓	✓
7	✓	✓	✓
8	✓	✓	
9	✓	✓	✓

5.5.6 RQ6: Enhancing GNNs with GraphPrior

Objectives: We investigate whether GraphPrior and the uncertainty-based metrics can select informative retraining subsets to improve the performance of a GNN model.

Experimental design: Following the prior research by Ma *et al.* [46], our retraining experiments are structured as follows. Firstly, we randomly partitioned the dataset into three sets: an initial training set, a candidate set, and a test set, with a ratio of 4:4:2. The candidate set was reserved exclusively for retraining purposes, while the test set was kept untouched for the purpose of evaluation. In the first round, we trained a GNN model using only the initial training set and computed its accuracy on the test set. We employed the best model obtained over the training epochs for the subsequent retraining process. In the second round, we incorporate an additional 10% of new inputs from the candidate set into the existing training set without replacement. The inputs selected for inclusion are those that are prioritized in the first 10% by the test prioritization approaches, namely GraphPrior and the compared techniques. Following Ma *et al.* [46], we retrain the GNN models by utilizing the complete augmented training set. This approach ensures that the old and new training data are treated equally. We repeat the retraining process for multiple rounds until the candidate set is empty. We kept the test data untouched during the retraining process. Moreover, we account for the randomness involved in the model training process and repeat all the experiments ten times to report the average

results (averaged over ten repetitions).

Results: Table 5.18 illustrates the average accuracy of GNN models after retraining with 10% to 100% prioritized test inputs. For each case, we highlight the approach with the highest effectiveness in grey to facilitate quick and easy interpretation of the results. **GraphPrior and the uncertainty-based test prioritization approaches outperform the random selection approach. However, the observed improvement is relatively small, indicating that GNN test prioritization approaches can guide the retraining of GNN models but with limited effect.** In Table 5.18, we observe that test prioritization methods, including GraphPrior and compared approaches, consistently demonstrate better performance across varying ratios of added data compared with the random selection. Furthermore, when incorporating prioritized tests exceeding 10% of the total, a significant majority of the test prioritization methods - specifically, 83.4% (10 out of 12) - outperform random selection in each case. However, the improvements achieved by these test prioritization methods compared to random selection are relatively small, with the highest increase being only 0.014. Additionally, Figure 5.4 visually depicts an example outcome of the retraining experiments conducted on the Cora dataset using the GCN model, showcasing a comparative evaluation of the performance of test prioritization approaches against random selection (indicated by the black line). As observed from the results, the test prioritization approaches demonstrate a better performance compared to random selection, but the improvement is visually slight.

One reason that leads to the effectiveness of GraphPrior and uncertainty-based test prioritization approaches being limited lies in their inadequate consideration of node importance (i.e., impact on other nodes in the dataset). In a GNN dataset, the complex interdependence among test inputs and their neighbors can lead to them having different importance. For example, nodes with greater connectivity can affect more of other nodes, making them relatively more critical. However, the current test prioritization approaches only focus on the ability of test inputs to reveal system bugs without regard to the importance of nodes. Although the selected test input by them can have a higher likelihood of misclassification, their importance within the dataset can be minor if they have a very small number of neighbors. Retraining such inputs would have less effect. Consequently, it is crucial to consider node importance in the selection of retraining data to achieve more effective outcomes.

GraphPrior achieved better effectiveness than the uncertainty-based test prioritization methods. In Table 5.18, we see that, when adding more than 20% (including 20%) test cases for retraining, the GraphPrior approaches perform the best in 100% cases. Figure 5.4 visually demonstrates that the GraphPrior approaches (solid line) perform better than the compared approaches (dotted line) in most cases.

Answer to RQ6: *GraphPrior and the uncertainty-based test prioritization approaches outperform the random selection approach. However, the observed improvement is relatively small, indicating that GNN test prioritization approaches can guide the retraining of GNN models but with limited effect. GraphPrior achieved better effectiveness than the uncertainty-based test prioritization methods.*

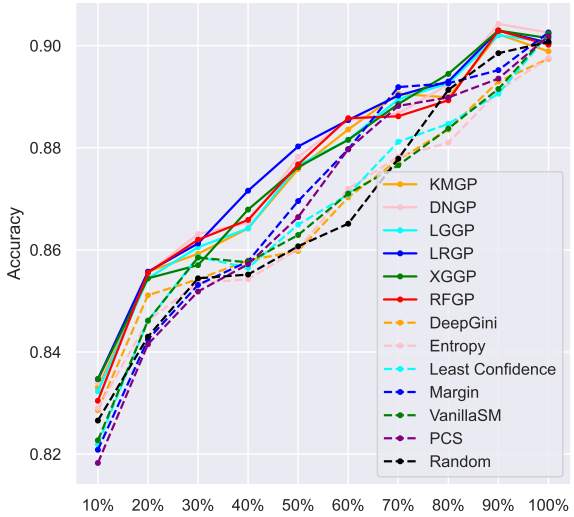


Figure 5.4: Enhancing the accuracy of the GNN with prioritized tests (Cora with GCN)

Table 5.18: The GNNs' average accuracy value after retraining with 10%~100% prioritized tests.

Approaches	Accuracy of percentage of datasets										Average
	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	
KMGP	0.787	0.810	0.825	0.834	0.844	0.854	0.861	0.867	0.874	0.878	0.844
DNGP	0.787	0.811	0.827	0.836	0.844	0.853	0.859	0.867	0.873	0.877	0.843
LGGP	0.787	0.812	0.825	0.835	0.845	0.852	0.861	0.868	0.873	0.877	0.844
LRGP	0.787	0.811	0.825	0.835	0.845	0.854	0.864	0.869	0.873	0.877	0.844
XGGP	0.788	0.811	0.824	0.834	0.845	0.853	0.861	0.867	0.873	0.877	0.843
RFGP	0.787	0.813	0.825	0.835	0.845	0.853	0.860	0.869	0.874	0.877	0.844
DeepGini	0.788	0.801	0.814	0.826	0.836	0.844	0.851	0.858	0.866	0.870	0.835
Entropy	0.789	0.801	0.816	0.829	0.836	0.845	0.852	0.858	0.866	0.872	0.837
LeastConfidence	0.789	0.802	0.816	0.828	0.836	0.846	0.853	0.860	0.866	0.872	0.837
Margin	0.788	0.801	0.818	0.827	0.837	0.845	0.853	0.861	0.867	0.872	0.837
VanillaSM	0.788	0.804	0.819	0.829	0.837	0.846	0.853	0.861	0.867	0.873	0.838
PCS	0.787	0.802	0.817	0.827	0.837	0.845	0.854	0.860	0.866	0.872	0.837
Random	0.789	0.799	0.814	0.825	0.834	0.843	0.853	0.860	0.866	0.872	0.836

5.6 Discussion

5.6.1 Generality of GraphPrior

Although the confidence-based test prioritization approaches demonstrate excellent effectiveness in traditional DNNs, they do not consider the interdependencies between test inputs, which are particularly crucial in GNN test prioritization. Our proposed GraphPrior leverages the mutation analysis of GNN models to perform GNN test input prioritization, which has been demonstrated effective on graph classification tasks through 604 carefully designed subjects. In fact, the scheme of GraphPrior, (i.e., modifying training parameters to mutate the GNN model for test prioritization) can also be generalized to other dimensions of GNN tasks, including graph-level and edge-level tasks. In the future, we will further verify the extension of GraphPrior from this perspective.

[*The applicability of GraphPrior on regression tasks*] In this section, we will also discuss the potential applicability of GraphPrior to regression tasks. Currently, the mutation rules and ranking models of GraphPrior are specifically designed for classification tasks. To extend GraphPrior to regression tasks, modifications to the mutation rules and ranking models would be required. If appropriate mutation rules can be identified for regression tasks and suitable ranking models can be designed, GraphPrior could also be applied to regression tasks.

5.6.2 Limitations of GraphPrior

[*Diversity of the prioritized data*] One limitation of GraphPrior lies in guaranteeing the diversity of selected bug data. This limitation is also noted in prior work on the uncertainty-based test prioritization approaches [3], which did not consider the diversity of bugs when prioritizing test inputs. Similarly, GraphPrior also does not aim for diversity in the prioritized tests. However, GraphPrior has demonstrated the ability to identify a significant majority of misclassified test inputs using a small ratio of prioritized test cases. Specifically, RFGP (i.e., the most effective GraphPrior approach) has been shown to detect over 80% misclassified tests by prioritizing only 40% of the test inputs. This highlights GraphPrior's ability to efficiently identify a large proportion of bugs using a small set of prioritized tests, even without explicitly ensuring bug diversity. While prioritizing diverse bugs can improve the overall quality of testing, prioritizing a significant majority of bugs can still be a practical strategy in situations where time and resources are limited. Therefore, GraphPrior's ability to efficiently identify a large proportion of bugs using a small set of prioritized tests can be particularly useful in scenarios where time and resources are constrained.

[*GraphPrior in active learning scenarios.*] Active learning [206] operates under the assumption that samples within a dataset have varying contributions to the improvement of the current model and aims to select the most informative samples for inclusion in the training set. Our investigation in RQ6 has demonstrated that GraphPrior and uncertainty-based metrics can be utilized to select informative retraining tests. However, the effectiveness of these approaches is limited. Specifically, despite the demonstrated success of uncertainty-based metrics such as DeepGini and margin in previous studies [3] [47] on DNNs, their effectiveness in the context of GNNs is slight. We explore potential reasons for this phenomenon.

One crucial reason for their limited effectiveness lies in their inadequate consideration of node importance, i.e., the impact that a node has on other nodes in

the graph dataset. In a GNN dataset, the complex interdependence among test inputs and their neighbors can result in differing levels of importance for different nodes. For instance, nodes with higher connectivity can be more influential and hence more critical. However, current test prioritization approaches only focus on the ability of test inputs to expose system bugs without taking into account the node importance. Although these approaches may identify inputs with a higher likelihood of misclassification, their importance within the dataset may be negligible if they have only a few neighbors. Retraining such inputs is, therefore, less effective.

Furthermore, we elaborate on the difference between GraphPrior and the existing active learning methods evaluated in our study. The active learning methods used for comparison in our paper are primarily uncertainty-based, aimed at datasets where each sample is independent of others. However, for graph datasets, these methods select retraining data without considering the interdependencies between nodes and also neglect the importance of nodes, merely selecting possibly-misclassifed nodes. In contrast, GraphPrior employs mutation analysis to identify test inputs that are more likely to be misclassified while considering the interdependencies between nodes during the mutation process. Despite this added consideration, GraphPrior's goal remains to select misclassified test inputs and does not explicitly consider node importance, leading to slight effectiveness as the uncertainty-based methods.

[*Generating mutants for large-scale GNN models*] In our experiments, which are based on our current model and datasets, the time cost of our retraining method (for generating mutants) is within an acceptable range. When dealing with large-scale GNN models, GraphPrior can require large computational resources, but it can remain feasible in situations where the cost of manual labeling outweighs the computational cost.

5.6.3 Threats to Validity

Threats to Internal Validity. The internal threats to validity mainly lie in the implementation of our proposed GraphPrior and the compared approaches. To reduce the threat, we implemented GraphPrior based on the widely used library PyTorch and adopted the implementations of the compared approaches published by their authors. Another internal threat lies in the randomness of the model training. To mitigate this threat and ensure the stability of our experimental results, we conducted a statistical analysis. Specifically, we repeated the training process ten times for both the original model and the mutated model and calculated the statistical significance of the experiments.

The selection of mutation rules in our study presents another internal threat to validity. Despite our best efforts to collect a comprehensive set of mutation rules, it is possible that other training parameters beyond our current knowledge could serve as mutation rules. To mitigate this threat, we selected mutation rules that can directly or indirectly affect node interdependence in the prediction process. The selection of parameter ranges for mutation rules is another internal threat that could affect the effectiveness of the rules. To mitigate this threat, we adopted a strategy in which we inverted the values of Boolean parameters, setting true to false and false to true. For integer and float parameters, we selected a range that introduces only slight changes to the original GNN model. Our experimental results demonstrated the effectiveness of GraphPrior, indicating that the mutation rules and selected parameter range are suitable for GNN test prioritization.

Threats to External Validity. The external threats to validity mainly lie in the GNN models under test and the testing datasets we used in our study. To mitigate this threat, we adopted a large number of subjects (pairs of model and dataset) in our study and leveraged different types of test inputs. We applied 8 graph adversarial attacks from public studies to generate adversarial test inputs and varied the attack level for more detailed evaluation. In the future, we will apply GraphPrior to more GNN models and test datasets with diversity.

5.7 Related Work

We present the related work in three aspects, which are test prioritization techniques, deep neural network testing, and mutation-based test prioritization for traditional software.

5.7.1 Test prioritization Techniques

In traditional software testing, test prioritization [92, 207, 142, 208, 145, 209, 210, 136] aims to find the ideal order of test cases to reveal system bugs earlier. Prioritizing test cases contributes to two critical constraints, time and budget for software testing, in order to detect more fault-revealing test cases in a limited time. Di Nardo *et al.* [210] conducted a case study of coverage-based prioritization strategies on real-world regression faults, evaluating the effectiveness of several test case prioritization techniques in bug detection. Rothermel *et al.* [136] presented and compared three types of test case prioritization techniques for regression testing that are based on test execution information. They demonstrated that each of the studied prioritization techniques increased the fault detection rate of the test suite. Henard *et al.* [142] conducted a comprehensive study to compare existing test prioritization approaches, finding that the difference between white-box [143, 211, 212, 92] and black-box strategies [144, 213, 214] are little. Chen *et al.* [145] proposed LET to prioritize test programs for compiler testing acceleration and demonstrated its effectiveness. LET works through two processes, the learning process to identify program features and predict the bug-revealing probability of a new test program and the scheduling process to prioritize test programs based on bug-revealing probabilities. Chen *et al.* [209] proposed to prioritize test programs based on the prediction information of the test coverage for compilers.

In terms of test prioritization for DNNs, Feng *et al.* [3] proposed the state-of-the-art approach, DeepGini, which identifies possibly-misclassified tests based on model uncertainty. DeepGini assumes a test is more likely to be mispredicted if the DNN outputs similar probabilities for each class. Byun *et al.* [147] evaluated several metrics that prioritize bug-revealing inputs based on the white-box measures of DNN's sentiment, including softmax confidence (i.e., predicted probability for output categories in DNNs that use softmax output layers), Bayesian uncertainty (i.e., the uncertainty of the prediction probability distributions for Bayesian Neural Networks), and input surprise (i.e., the distance of the neuron activation pattern between a test input and the training data). Wang *et al.* [2] proposed PRIMA to prioritize test inputs for DNNs via intelligent mutation analysis. PRIMA further improves DNN test prioritization in two main aspects. First, PRIMA can be applied not only to classification modes but also to regression models. Second, PRIMA can deal with the case in which test inputs are generated from adversarial input generation approaches [215] that can make the probability of the wrong class larger. Furthermore,

some data selection approaches [172] are also proposed to detect possibly-misclassified tests for DNNs. Despite its effectiveness in DNN test prioritization, the PRIMA approach cannot be directly applied to GNNs. This is because PRIMA's mutation operators are not adapted to graph-structured data and GNN models.

More specifically, GNN models operate on graph-structured data, where nodes and edges represent entities and their relationships. Conversely, the input mutation rules of PRIMA were designed for independent test samples, rendering them unsuitable for GNNs. Moreover, GNNs incorporate unique graph operations and aggregation mechanisms, including graph convolution operations and message passing mechanisms. PRIMA's model mutation rules are not applicable to the graph-level mechanisms employed by GNNs. As such, GNNs require specialized test prioritization techniques, such as GraphPrior, which leverages the properties of GNN models in its mutation analysis for test prioritization. More specifically, to address the limitations of PRIMA, GraphPrior introduces mutation rules that are designed based on the graph operations and aggregation mechanisms of GNNs. These rules can directly or indirectly impact message passing. Consequently, GraphPrior enables prioritizing tests for graph-structured data.

5.7.2 Deep Neural Network Testing

Besides test input prioritization, some test selection approaches have also been proposed to improve the efficiency of DNN testing. Test selection aims to precisely estimate the accuracy of the whole set by only labeling the set of selected test inputs. In this way, the labeling cost for DNN testing is reduced. Li *et al.* [36] proposed CES (Cross Entropy-based Sampling) and CSS (Confidence-based Stratified Sampling) to select a small group of representative test inputs to estimate the accuracy of the whole testing set. CES minimizes the cross-entropy between the selected set and the entire test set to ensure that the distribution of the selected test set is similar to the original test set. CSS leverages the confidence features of test inputs to guarantee the similarity between the selected test set and the entire test set. Chen *et al.* [35] proposed PACE (Practical Accuracy Estimation), which selects test inputs practically based on clustering, prototype selection, and adaptive random testing. Pace first clusters all the test inputs into different groups based on their testing capabilities. Then, Pace utilizes the MMD-critic algorithm [37] to select prototypes from each group. For test inputs not in any group, Pace leverages adaptive random testing to select tests from them. Compared to the aforementioned research, our work focus on test prioritization, which ranks all the test inputs without discarding any test input. In this way, testers or developers can find the test inputs that reveal bugs earlier.

In addition to improving the efficiency of DNN testing, several existing studies [4, 5, 48, 49, 151, 50] have focused on measuring the adequacy of DNNs. Pei *et al.* [4] proposed a metric of neuron coverage to evaluate how adequate a test set covers the logic of a DNN model. Based on this metric, they proposed a white-box framework for testing DNNs. In the following study, Ma *et al.* [5] proposed DeepGauge, a set of DNN testing coverage criteria to measure the test adequacy of DNNs. DeepGauge also considers neuron coverage to be a good indicator of the effectiveness of a test input. Based on the basic neuron coverage metric, they proposed new metrics with different granularities to differentiate adversarial attacks from legit test data. Kim *et al.* [49] proposed the surprise adequacy for testing of DL models, which identify how effective a test input by measuring its surprise with respect to the training set.

More specifically, the surprise of a test input refers to the difference in the activation value of neurons in the face of this new test.

5.7.3 Mutation Testing for DNNs

Several existing studies have explored the use of mutation testing for DNNs and developed different mutation operators and frameworks. Ma *et al.* [48] propose DeepMutation to measure the quality of test data for DL systems based on mutation testing. To this end, they design a set of source-level and model-level mutation operators to inject faults into the training data, training programs, and DL models. The quality of test data is evaluated by analyzing the extent to which the injected faults can be detected. The work by Ma *et al.* was later extended into a mutation testing tool for DL systems named DeepMutation++ [151], which proposed a set of new mutation operators for feed-forward neural networks (FNNs) and Recurrent Neural Networks (RNNs) and can dynamically mutate run-time states of an RNN. Humbatova *et al.* [146] proposed DeepCrime, which is the first mutation testing tool that implements a set of DL mutation operators based on real DL faults. Shen *et al.* [216] proposed MuNN, a mutation analysis method for neural networks. MuNN defined five mutation operators based on the characteristics of neural networks. The results reveal that mutation analysis has strong domain characteristics, indicating the need for domain mutation operators to enhance the analysis, and that new mutation mechanisms are required for deep neural networks.

The above studies in mutation testing have focused on traditional DNNs, which are typically evaluated on datasets with independent samples. However, the mutation rules employed in these studies do not account for the interdependence among test inputs, which is a crucial factor to consider in the context of GNN testing. In contrast, the mutation rules of GraphPrior are designed to impact the message passing mechanism in the GNN prediction process. In the mutated GNN model, the way nodes acquire information from their neighboring nodes differs slightly from that of the original GNN model. The mutation features generated based on these mutation rules are fed into ranking models to predict the likelihood of a test input being misclassified by the GNN model.

5.7.4 Mutation-based Test Prioritization for Traditional Software

In traditional software testing, mutation testing is a well-established technique to evaluate the quality of test sets. Mutation-based test prioritization focuses on prioritizing test cases based on their ability to detect mutants. The key idea is that test cases that can detect mutants are likely to be more effective at finding real faults in the code and, therefore, should be given higher priority. Several mutation-based approaches [141, 150] have been proposed. Lou *et al.* [141] proposed a test-case prioritization approach based on the fault detection capability of test cases. They introduced two models to calculate the fault detection capability: the statistics-based model and the probability-based model. Based on the experimental study, they found that the statistics-based model outperforms all the approaches. Shin *et al.* [150] proposed a test case prioritization technique guided by the diversity-aware mutation adequacy criterion and empirically evaluated the effectiveness of mutation-based prioritization techniques with large-scale developer-written test cases. Papadakis *et al.* [64] proposed mutating Combinatorial Interaction Testing models and using them to prioritize tests based on their ability to kill mutants and showed that the number

of model-based mutants that are killed yields a strong correlation to code-level faults revealed by the test cases. The aforementioned DNN-oriented approaches consider each test input independent of each other, while in a graph dataset, there are usually complex connections between test inputs. Our proposed GraphPrior specifically targets GNNs and utilizes several mutation rules to generate GNN mutants for test prioritization. Moreover, to better leverage the mutation results, we adopt several ranking models [14, 112, 69] that can learn to predict the probability of a test input to be misclassified.

5.8 Conclusion

To improve the efficiency of GNN testing, we aim to prioritize possibly-misclassified test inputs to reveal GNN bugs earlier. However, a crucial limitation of existing test prioritization approaches is that, when applying to GNNs, they do not take into account the interdependence between test inputs (nodes). In this paper, we propose GraphPrior, a set of test prioritization approaches specifically for GNN testing. GraphPrior assumed that a test input is more likely to be misclassified if it can kill many mutated models. Based on it, GraphPrior leveraged carefully designed mutation rules to generate mutated models for GNNs. Subsequently, GraphPrior obtained the mutation results of test inputs based on the execution of the mutated models. GraphPrior utilized the mutation results in two ways, namely, killing-based and feature-based methods. In the process of scoring a test, killing-based methods considered each mutated model equally important, while feature-based methods learned different importance for each mutated model through ranking models. Finally, GraphPrior ranked all the test inputs based on their scores. We conducted an extensive study to evaluate the effectiveness of GraphPrior approaches on 604 subjects, comparing them with existing approaches that could detect possibly-misclassified test inputs. The experimental results demonstrate the effectiveness of GraphPrior. In terms of APFD, the killing-based GraphPrior approach, KMGP, exceeds the compared approaches (i.e., DeepGini, margin, Vanilla Softmax, PCS, Entropy, least confidence and random selection) by 0.034~0.248 on average. Furthermore, RFGP (i.e., the feature-based GraphPrior approach) exhibited better performance compared to other GraphPrior approaches. Specifically, RFGP outperforms the uncertainty-based test prioritization approaches against different adversarial attacks, with the average improvement of 2.95%~46.69%.

6 LongTest: Test Prioritization for Long Text Files

In this chapter, we propose LongTest, a novel test prioritization approach specifically designed for long text files. LongTest addresses critical gaps in the literature: 1) Confidence-based methods fail to leverage the rich semantic content of the test inputs (long text files) for test prioritization; 2) mutation-based techniques are specifically designed for short texts, introducing small mutations that have negligible impacts on long text files; and 3) coverage-based methods are computationally expensive and less effective. To this end, LongTest introduces ***1) an embedding generation mechanism*** *specifically designed to enhance the capture of information from the entire long text file, and* ***2) contrastive learning*** *that enables more effective differentiation between misclassified and correctly classified samples, which optimizes the test prioritization process. The experimental results demonstrate the effectiveness of LongTest, outperforming all the compared test prioritization approaches.*

This chapter is based on the work in the following research paper:

- Xueqi Dang, Yinghua Li, Wendkuuni C. Ouédraogo, Maxime Cordy, Mike Papadakis, Jacques Klein, Tegawendé F. Bissyandé, Yves Le Traon. LongTest: Test Prioritization for Long Text Files. Under Review in ACM Transactions on Software Engineering and Methodology (TOSEM), 2024.

Contents

6.1 Introduction

Text classification is a foundational problem in natural language processing (NLP) [217], which focuses on assigning pre-defined labels to text files. While traditional research [217, 218] has primarily targeted short texts (e.g., social media posts, emails, and product reviews), the increasing complexity of real-world applications demands more advanced methods capable of analyzing long text files, such as legal documents, scientific literature, and technical reports.

Long texts present unique challenges due to their extended length, complex hierarchical structures, and diverse semantic content, distinguishing them significantly from short texts [28]. The classification of long texts holds critical importance across various domains, such as medical report classification [219], legal document categorization [220], and research topic classification [221, 222]. For example, in the medical domain [223], classification models can be utilized to categorize patients' medical history records (which are long text files) into different disease types, thereby facilitating accurate diagnoses and improving treatment planning.

However, in such critical scenarios, misclassifications can lead to severe consequences. For example, if a patient's record is misclassified, it could result in misdiagnosis or inappropriate treatment, delaying the correct medical process and potentially worsening the patient's condition. Therefore, ensuring the accuracy and reliability of such long-text classification systems is essential. Testing is a fundamental practice for ensuring the quality of DNN-based systems [3]. However, labeling test inputs to verify the correctness of classification results can be costly [35]. This challenge has also extended to the field of long text classification due to three main reasons: 1) Manual annotation remains a mainstream method for labeling, making the process labor-intensive and time-consuming. 2) Test sets for long text classification can be large-scale, increasing the labeling workload. 3) Domain-specific knowledge is frequently required in certain fields to accurately label long text files. For example, annotating medical history records can require expertise from medical professionals, further increasing the labelling cost.

To alleviate the labelling cost problem, one intuitive and effective approach is test input prioritization [3, 7, 2], which focuses on identifying and prioritizing test inputs that are more likely to be misclassified by the DNN model, as these inputs are more likely to expose faults in the model. This approach enables the detection of more fault-revealing test inputs within a limited timeframe. Labeling these inputs earlier accelerates the debugging process and thus improves the efficiency of DNN testing.

In the literature, several test prioritization methods have been proposed to enhance the efficiency of DNN testing. These approaches can be broadly categorized into three classes: coverage-based [4, 5], confidence-based [3, 7], and mutation-based approaches [2]. Coverage-based methods adapt traditional software testing techniques to DNN testing by leveraging neuron coverage metrics for test prioritization. Confidence-based methods focus on prioritizing test inputs where the model exhibits lower confidence in its predictions. For example, DeepGini [3] uses the Gini score to quantify the model's prediction confidence and prioritizes inputs with higher uncertainty. Mutation-based methods [2] introduce novel mutation operators and utilize mutation results to guide test prioritization.

However, although these approaches have demonstrated effectiveness in certain

cases, they suffer from the following limitations when applied to the context of long text files:

- Confidence-based approaches rely solely on the output probabilities generated by the final layer of the model, which represent a compressed summary of the model's confidence in its predictions. They do not utilize the information contained within the original input data, especially the rich semantic and hierarchical structure of long text files, which, as a result, limits their effectiveness in prioritizing long text files.
- Mutation-based approaches are typically designed for short text. Their proposed mutation operators usually introduce small-scale mutations, such as altering a few characters within a word. However, long text typically consists of extended content and numerous words, making the impact of such slight mutations negligible, which makes mutation-based methods unsuitable for test prioritization in this context.
- Coverage-based test prioritization methods have been demonstrated to be less effective and more computationally expensive compared to confidence-based methods [3].

In this paper, we propose LongTest (**Long** text file-oriented **Test** Input Prioritization), a learning-based test input prioritization approach specifically designed for prioritizing long text files. The core idea is that we utilize *contrastive learning* to bring misclassified tests closer together in the space while pushing misclassified and correctly classified tests farther apart. This enables better differentiation between misclassified and correctly classified tests, thereby achieving more effective test prioritization.

Specifically, LongTest performs test prioritization based on four steps: 1) **Text Splitting and Transforming.** For each test input (a long text file) in the test set, LongTest divides it into smaller chunks, converts each chunk into an embedding, and concatenates all chunk embeddings to generate a final comprehensive embedding vector. This process aims to enhance the capture of information from the entire long text, particularly because embedding algorithms can have input token limitations (i.e., The embedding algorithms are constrained by the input text length and cannot capture information exceeding this limit). **2) Dimensionality Reduction.** LongTest applies Principal Component Analysis (PCA) [224] to reduce the dimensionality of the embedding vector of each test, aiming to decrease the overall runtime and improve efficiency while retaining the essential characteristics of the original data. **3) Contrastive Learning.** LongTest utilizes contrastive learning to bring misclassified tests closer in space while pushing misclassified and correctly classified tests farther apart. As a result, after this step, the original embedding vector of each test is transformed into a contrastive vector. **4) Ranking.** LongTest employs a trained classification model to estimate the misclassification probability for each test and ranks all tests based on these probability values.

LongTest has the following notable advantages when applied to prioritize long text files:

- **Leveraging Long Text Characteristics** Unlike confidence-based methods that solely rely on output probabilities from the model's final layer, LongTest incorporates rich semantic information from the entire long text file. Specifically, LongTest processes long text files by splitting them into chunks, generating embedding vectors, and applying Principal Component Analysis (PCA) for dimensionality reduction. This process preserves essential information of long text files while

reducing the total running time and improving efficiency.

- **Enhancing Differentiation via Contrastive Learning** LongTest leverages contrastive learning to bring misclassified test samples closer together in the feature space while pushing misclassified and correctly classified samples further apart. This approach enables LongTest to better distinguish between misclassified and correctly classified samples, thereby improving the prioritization effectiveness.
- **Specifically Designed for Long Text** LongTest is specifically designed for prioritizing long text files, fully accounting for the characteristics of their extended length. Traditional mutation-based prioritization methods are designed for short texts, with mutation operators proposed specifically for them, making such methods unsuitable for long text files.

LongTest demonstrates broad application across various scenarios. One typical application is the classification of medical history records. In this context, classification models are used to categorize patients' medical history records into different disease types. However, incorrect classification can lead to severe consequences. For instance, if a patient's record is misclassified, it can result in misdiagnosis or inappropriate treatment, delaying the correct medical process and potentially worsening the patient's condition. In such cases, LongTest can effectively identify and prioritize medical history records that are more likely to be misclassified by the model. These potentially misclassified records can then be prioritized for manual review by medical experts, thereby reducing the risks associated with incorrect classification. Additionally, developers can conduct an in-depth analysis of the characteristics of misclassified records to analyze the causes of prediction errors, thus further optimizing and improving the classification model.

To assess the effectiveness of LongTest, we performed a comprehensive experimental evaluation using 45 subjects comprising three datasets and 15 DNN models. We compared LongTest against several existing test prioritization approaches that have demonstrated effectiveness in prior research [3, 7]. The experimental results exhibit that LongTest outperforms all existing test prioritization methods, with an average improvement ranging from 14.28% to 70.86%. To facilitate further research, we have made our dataset, results, and tools publicly available on Github[1].

Our work has the following major contributions:

- **Approach** We proposed LongTest, a test prioritization approach specifically designed for prioritizing long text files. To this end, we designed an embedding generation method tailored for long texts and applied contrastive learning to enhance the effectiveness of test prioritization.
- **Study** We evaluated LongTest using 45 subjects, including three datasets and 15 DNN models. The experimental results demonstrate that LongTest outperforms all the compared test prioritization approaches, with an average improvement of 14.28%~70.86%.
- **Performance Analysis** We investigate the impact of various parameters and components of LongTest on its performance, including the number of chunks (RQ2), embedding models (RQ3), dimensionality (RQ4), and main parameters (RQ5). Additionally, we conduct an ablation study (RQ6) to analyze the contribution of each individual component to the effectiveness of LongTest.

[1]https://github.com/yinghuali/LongTest

6.2 Background

6.2.1 Deep Neural Networks

Deep Neural Networks (DNNs) consist of multiple layers, each containing numerous interconnected units known as neurons [21]. These neurons are connected by links, each link assigned a specific weight that is optimized during the model's training process. The training data guides the adjustment of these weights, enabling the network to learn and generalize patterns across various tasks. Text classification [24] is a common application of DNNs, focusing on categorizing textual data into predefined classes based on its content. It is widely used in fields like sentiment analysis [25], spam detection [26, 27], and medical record analysis [225]. Text classification tasks can vary depending on text length and complexity. Short texts, such as social media posts or product reviews, are typically brief, with a small number of characters and straightforward content. Conversely, long texts, including news articles or legal documents, contain extensive information with complex structures, often spanning multiple topics and sentiments. Compared to short text classification, long text classification presents more challenges due to the need to capture information spread across extensive content.

6.2.2 Contrastive Learning

Contrastive learning [45] learns effective data representations by pulling similar samples closer in feature space and pushing dissimilar samples apart. In recent years, contrastive learning has shown notable performance across various fields, especially with applications in image, text, and multi-modal data [226]. The primary objective of contrastive learning is to identify a parametric function $f_\theta \colon \mathbb{R}^D \to \mathbb{R}^d$ that maps an input image $\mathbf{x} \in \mathbb{R}^D$ to a feature representation $\mathbf{z} = f_\theta(\mathbf{x}) \in \mathbb{R}^d$. This representation $\mathbf{z}$ in the feature space is designed to capture the semantic similarities between input samples. To accomplish this objective, contrastive learning leverages a **contrastive loss** function to optimize the network f_θ. The contrastive loss is designed to draw semantically similar samples closer together in the feature space while pushing apart dissimilar samples. This method enables the model to learn discriminative features that effectively distinguish between different categories (between positive samples and negative samples).

In our study, we propose the LongTest method, which leverages contrastive learning to enhance test prioritization effectiveness on long text files. Specifically, we define misclassified tests (those incorrectly predicted by the DNN model) as positive samples and correctly classified tests as negative samples. By training the contrastive learning model in this manner, LongTest aims to better distinguish between misclassified and correctly classified cases, thereby improving the model's ability to predict the probability of misclassification for a given test input. This can ultimately enhance the effectiveness of test prioritization.

6.2.3 Test Input Prioritization for DNNs

Test prioritization aims to identify and prioritize potentially misclassified test cases earlier in the testing process [3, 227, 228, 33]. Given a model M to test and a test suite T, the goal of test input prioritization is to systematically rank the test cases in T so that, when testing is halted at a certain point, the executed test cases from the prioritized set can reveal as many faults as possible. This approach enables

the identification of bug-revealing test inputs within a limited timeframe, facilitating earlier debugging and improving the efficiency of DNN testing.

The existing DNN test prioritization approaches can be broadly classified into three main categories: coverage-based [4, 5], confidence-based [3, 7, 229], and mutation-based approaches [2, 34]. Coverage-based approaches adapt traditional software testing methods to DNN testing by leveraging neuron coverage metrics for prioritizing test inputs. Confidence-based approaches, on the other hand, prioritize test cases based on the model's prediction confidence, under the assumption that test cases with lower confidence are more likely to be misclassified. For example, DeepGini [3] calculates the Gini score to quantify the model's confidence for each test case and ranks the test cases within the test set based on these scores. Mutation-based approaches [2] introduce novel mutation operators, including model and input mutation operators, and utilize the mutation results for test prioritization.

However, although the above approaches have demonstrated effectiveness in certain cases, they face limitations when applied to long text files: 1) Coverage-based methods have been shown to be less effective and more computationally expensive compared to confidence-based approaches [3]; 2) Confidence-based approaches depend solely on output probabilities from the model's final layer. They do not use the rich semantic and hierarchical structure of test inputs (long text files) for test prioritization, limiting their effectiveness. 3) Mutation-based methods, typically designed for short text, focus on small-scale mutations, such as modifying a few characters of a word. These small mutations have negligible impact on long text files due to their extended length, making mutation-based approaches unsuitable for such cases. To address the above limitations, in this paper, we propose LongTest, which leverages the unique characteristics of long text files for test prioritization. Section 6.5.1 demonstrates the effectiveness of LongTest.

6.3 Approach

6.3.1 Overview

In this paper, we propose a novel test prioritization approach called LongTest. LongTest is specifically designed for prioritizing long text files. Specifically, the input to LongTest is an unlabelled test set T and a text classification model M to be evaluated. LongTest will output a prioritized test set T', where tests that are more likely to be misclassified are prioritized higher. An overview of LongTest is presented in Figure 6.1. As shown in Figure 6.1, the construction of the LongTest approach is divided into four steps. The specific details of each step can be found in the following sections (cf. Section 6.3.2 to Section 6.3.5).

❶ **Step1: Text Preprocessing and Dimensionality Reduction** Given the training set R, for each long text-type sample $l \in R$, we first preprocess it. This preprocessing includes three key steps: 1) splitting the long text l into several chunks, 2) transforming each chunk into embeddings, and 3) concatenating the embeddings of all chunks to obtain a final embedding vector, denoted as V_l. Next, for each sample l, we perform Principal Component Analysis (PCA) [224] to reduce the dimensionality of its vector V_l and obtain a reduced-dimensional vector V_l^{PCA}.

❷ **Step2: Constructing Positive and Negative Pairs** After obtaining the final dimension-reduced embedding vector of each sample in R, we use these vectors to

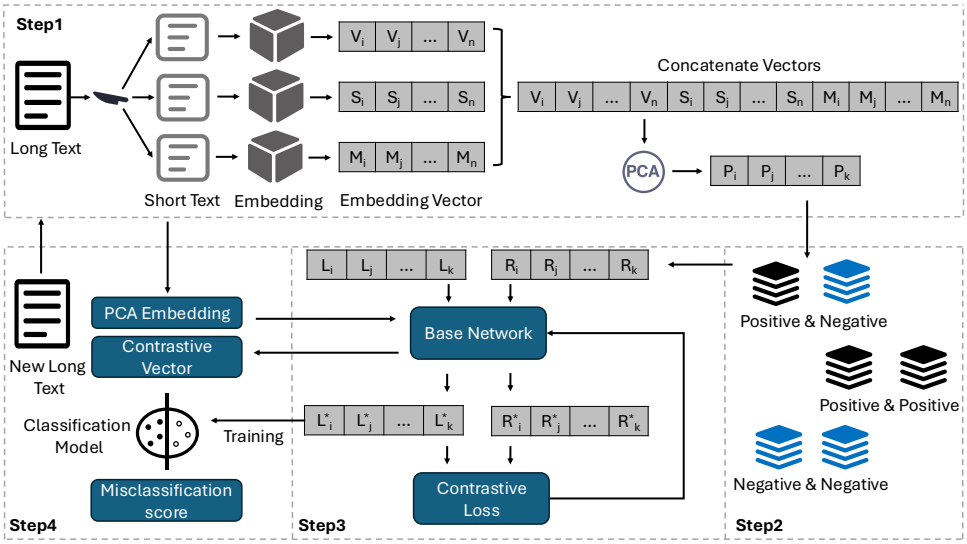


Figure 6.1: Overview of LongTest

create pairs (e.g., positive & negative pairs). Here, "positive" represents samples that are misclassified by the model, while "negative" represents samples that are correctly classified by the model. Since we know the ground truth for each sample in R, we can simply input each sample in R into the text classification model M, compare its prediction with the ground truth, and annotate each sample as misclassified (positive) or correctly classified (negative). The generated pairs are then passed to the Base Network in Step 3 to train the contrastive learning model.

❸ **Step3: Training Contrastive Learning Model** In this step, we use the paired data from the previous step to train the contrastive learning model. After training, this contrastive learning model will bring the embedding vectors of misclassified tests closer to each other in space and make the embedding vectors of correctly classified tests closer to each other. Additionally, it will make the misclassified tests farther away from the correctly classified tests. This is intended to enable the subsequent classification model to better predict whether a test will be misclassified or correctly classified based on its vector.

Specifically, after training, if we input the embedding vector V_t^{PCA} of a new test t into the contrastive learning model, it will output a transformed contrastive vector V_t^{cont}. The contrastive vector V_t^{cont} can better reflect how likely a test t will be misclassified.

❹ **Step4: Training Classification Model for Prioritization** In the final step, we trained a classification model for test prioritization. This model predicts whether a sample will be misclassified based on its contrastive vector, where a misclassified sample is labelled as "1" and a correctly classified sample as "0". Once the model was trained, we made some adjustments to it. Specifically, given an input (the contrastive vector of a test), we make the model output the intermediate value instead of the binary label (0 or 1). This value is a probability value indicating the likelihood that the test will be misclassified. We refer to it as the "misclassification score". The higher this score, the more likely the test will be misclassified.

After the construction of LongTest, given a test set T and model M to be evaluated, LongTest will predict the misclassification score of each test $t \in T$ to rank them. Tests that have high misclassification scores will be prioritized higher.

6.3.2 Step 1: Text Preprocessing and Dimensionality Reduction

Given the training set R and the text classification model M to be evaluated, for each long-text sample $l \in R$, we first performed text preprocessing and dimensionality reduction on it. In the following, we provide a detailed explanation of each specific procedure, along with the relevant formulas.

6.3.2.1 Chunk-based Text Splitting

Given the long text sample $l \in R$, we split l into several chunks, as described in Formula 6.1.

$$l = [c_1, c_2, \ldots, c_n] \tag{6.1}$$

where c_i represents the i_{th} chunk. n is the total number of chunks.

Rationale: The reason we perform text splitting is that embedding models typically have input token limitations, meaning that they are constrained by the length of the input text and cannot process information beyond this limit. Therefore, if the long text l exceeds the token limit, the portion of the text exceeding this limit will be lost when converting the text into an embedding vector, resulting in information loss. Through chunk-based splitting, we aim to enhance the capture of information from the entire long text file.

Specifically, in RQ2 (cf. Section 6.5.2), we investigate the impact of the number of chunks on the effectiveness of LongTest.

6.3.2.2 Transforming Text into Embeddings

Next, for each long text sample $l \in R$, we transform the obtained chunks $l = [c_1, c_2, \ldots, c_n]$ individually into embeddings using an embedding algorithm [230]. Then, we concatenate all the embeddings of the chunks of l to generate a final embedding vector V_l. This process is represented as Formula 6.2.

$$V_l = [\text{Embedding}\,(c_1)\,, \text{Embedding}\,(c_2)\,, \ldots, \text{Embedding}\,(c_n)] \tag{6.2}$$

where Embedding(c_i) denotes the embedding of the chunk c_i. V_l refers to the embedding vector of the long text sample l.

Rationale: The reason we transform each long text sample $l \in R$ into an embedding vector is that text is unstructured data, which models cannot directly understand. Embeddings convert text into fixed-dimensional vector representations, preserving the semantic information of the text and facilitating subsequent analysis. Through embeddings, the text content of $l \in R$ can be used to train the classification model in the following step (which is used to predict how likely a long text sample will be misclassified).

Specifically, in RQ3, as detailed in Section 6.5.3, we investigate the impact of different embedding algorithms on the effectiveness of LongTest.

6.3.2.3 Dimensionality Reduction with PCA

In the previous step, for each long text sample l in the training set R, we obtained its embedding vector V_l. In this step, we perform dimensionality reduction on this embedding vector using the Principal Component Analysis (PCA) algorithm [224], aiming to reduce its dimensionality to decrease the overall runtime while preserving the essential semantic information of the long text. This process can be represented as Formula 6.3.

$$V_l^{PCA} = \text{PCA}(V_l) \tag{6.3}$$

where V_l is the original embedding vector of each long text sample $l \in R$. PCA() denotes the Principal Component Analysis algorithm used for dimensionality reduction. V_l^{PCA} is the resulting lower-dimensional embedding of l.

Rationale: Dimension reduction aims to reduce the overall runtime of LongTest and enhance the efficiency of test prioritization while preserving the features of the original data [231]. High dimensionality can lead to the **curse of dimensionality**, which causes computational inefficiency and the risk of overfitting due to sparse data distribution in high-dimensional spaces.

6.3.3 Step 2: Constructing Positive and Negative Pairs

In this step, we construct positive and negative pairs for training the contrastive learning model. The specific steps are as follows:

❶ First, for each long text sample $l \in R$, we use the text classification model M to be evaluated to predict the label of l, obtaining the prediction result $\hat{y}_l$.

❷ Given that the ground truth of l is y_l, we compare $\hat{y}_l$ with y_l to determine whether l is misclassified by the model. If l is misclassified, we label it as "positive" (+); otherwise, we label it as "negative" (−). This process is presented in Formula 6.4

$$z_l = \begin{cases} +1, & \text{if } \hat{y}_l \neq y_l \quad \text{(misclassified, labeled as "positive")} \\ -1, & \text{if } \hat{y}_l = y_l \quad \text{(correctly classified, labeled as "negative")} \end{cases} \tag{6.4}$$

where z_l represents whether sample l is misclassified by the model, with $z_l = +1$ indicating that the sample is misclassified (labeled as "positive") and $z_l = -1$ indicating that the sample is correctly classified (labeled as "negative"). Here, $\hat{y}_l$ is the predicted label generated by the text classification model M, and y_l is the ground truth label of the sample l. If $\hat{y}_l \neq y_l$, it corresponds to misclassification. If $\hat{y}_l = y_l$, it corresponds to correct classification.

❸ After labeling all the samples in the training R as "positive" or "negative", we construct positive and negative sample pairs. Specifically, we construct three types of sample pairs in total: positive-negative, positive-positive, and negative-negative, which are presented as follows. These constructed pairs will be used in subsequent steps to train the contrastive learning model.
In the following, a and b represent two arbitrary data samples from the training dataset R.

- **Positive-Negative Pairs** ($\mathcal{P}_{PN}$) The positive-negative pair set $\mathcal{P}_{PN}$ includes pairs where $a, b \in R$, one sample has a positive label, and the other has a

negative label:

$$\mathcal{P}_{PN} = \{(\mathbf{v}_a^{PCA}, \mathbf{v}_b^{PCA}) \mid a, b \in R, z_a = +1, z_b = -1, a \neq b\}.$$

– **Positive-Positive Pairs** ($\mathcal{P}_{PP}$) The positive-positive pair set $\mathcal{P}_{PP}$ includes pairs where $a, b \in R$ and both samples have positive labels:

$$\mathcal{P}_{PP} = \{(\mathbf{v}_a^{PCA}, \mathbf{v}_b^{PCA}) \mid a, b \in R, z_a = +1, z_b = +1, a \neq b\}.$$

– **Negative-Negative Pairs** ($\mathcal{P}_{NN}$) The negative-negative pair set $\mathcal{P}_{NN}$ includes pairs where $a, b \in R$ and both samples have negative labels:

$$\mathcal{P}_{NN} = \{(\mathbf{v}_a^{PCA}, \mathbf{v}_b^{PCA}) \mid a, b \in R, z_a = -1, z_b = -1, a \neq b\}.$$

where $\mathbf{v}_a^{PCA}$ and $\mathbf{v}_b^{PCA}$ refer to the PCA-reduced embedding vectors for samples a and b, respectively. z_a and z_b represent the labels (positive or negative) of samples a and b, respectively. $z_a = +1$ indicates that the sample a has a positive label, and $z_b = -1$ indicates that the sample b has a negative label. The condition $a \neq b$ ensures that a and b are not the same sample.

❹ Finally, we obtained the complete set of sample pairs:

$$\mathcal{P} = \mathcal{P}_{PN} \cup \mathcal{P}_{PP} \cup \mathcal{P}_{NN},$$

where $\mathcal{P}$ refers to the complete set of sample pairs, which is the union of $\mathcal{P}_{PN}$, $\mathcal{P}_{PP}$, and $\mathcal{P}_{NN}$. $\mathcal{P}_{PN}$ refers to the set of Positive-Negative Pairs. $\mathcal{P}_{PP}$ refers to the set of Positive-Positive Pairs. $\mathcal{P}_{NN}$ refers to the set of Negative-Negative Pairs.

6.3.4 Step 3: Training Contrastive Learning Model

Using the sample pairs $\mathcal{P}$ from the above step, we train the contrastive learning model. The contrastive loss function works by bringing same-type pairs (Positive-Positive Pairs, Negative-Negative Pairs) closer in the space while pushing different-type pairs (Positive-Negative Pairs) farther apart in the space. This brings misclassified tests ("positive") closer together in space while pushing misclassified ("positive") and correctly classified tests ("negative") farther apart. This enables better differentiation between misclassified and correctly classified tests, thereby achieving more effective test prioritization.

After training, we obtain the base network of contrastive learning, which can be used to transform embedding vectors of a test input. Specifically, if we input the dimensionality-reduced embedding vector V_t^{PCA} of a long-text test t into the trained base network, a contrastive vector will be generated based on it. This process is mathematically represented in Formula 6.5.

$$\mathbf{v}_t^{\text{cont}} = f_\theta\left(\mathbf{v}_t^{\text{PCA}}\right) \tag{6.5}$$

where $\mathbf{v}_t^{\text{PCA}}$ denotes the input vector, representing the dimensionality-reduced embedding of the long-text test t. f_θ denotes the base network, a transformation function parameterized by θ, which is learned during the training phase of the contrastive learning model. $\mathbf{v}_t^{\text{cont}}$ denotes the output vector, referred to as the "contrastive vector", which maps the long-text test in a space optimized for distinguishing between misclassified and correctly classified tests.

6.3.5 Step 4: Training Classification Model for Prioritization

In the final step, we trained a Random Forest model [232] to classify between misclassified tests and correctly classified tests. We chose to use the Random Forest model because it is faster compared to other classification models, such as XGBoost [8] and LightGBM [69], while maintaining high effectiveness. Specifically, we built a new training dataset R_{RF} based on the original training dataset R for training the Random Forest model. Below, we describe how the training dataset is constructed:

- **Feature Extraction** Based on the steps described from Section 6.3.2 to Section 6.3.4, we transformed each long-text sample $l \in R$ into a contrastive vector $\mathbf{v}_l^{\text{cont}}$ through text preprocessing, dimensionality reduction, and enhancement by the contrastive learning model.
- **Label Assignment** We label each sample $l \in R$ as either misclassified or correctly classified by the model by comparing the model's prediction result with its ground truth. If the sample is misclassified by the model, we label it as 1; otherwise, we label it as 0.
- **Training Dataset Construction** For each $l \in R$, we use its contrastive vector $\mathbf{v}_l^{\text{cont}}$ as features and whether it is misclassified (1 for misclassified and 0 for correctly classified) as labels to construct the new training dataset R_{RF} for training the Random Forest model.

Once the Random Forest model was trained, we made slight adjustments to it so that, when inputting a test t, it could output the probability that t would be misclassified by the model. Specifically, we configured the Random Forest model to output the intermediate prediction value. This intermediate value represents the probability that the model predicts the test as belonging to class 1 ("misclassified") and, thereby, can be used to estimate the likelihood that a sample will be misclassified. We used this value as the **misclassification score** to rank all the tests.

6.3.6 Usage of LongTest

In this section, we present how LongTest performs test prioritization on an unordered, unlabeled long-text test set $T = \{t_1, t_2, \ldots, t_n\}$, given the text classification model M to be evaluated.

❶ For each long-text test sample t_i in T, LongTest processes it using the text preprocessing and dimensionality reduction steps described in Step 1 (cf. Section 6.3.2), obtaining the reduced-dimensional embedding for each test. The resulting set of embeddings is represented as:

$$\mathcal{T}_{\text{emb}} = \left\{ \mathbf{v}_{t_1}^{\text{PCA}}, \mathbf{v}_{t_2}^{\text{PCA}}, \ldots, \mathbf{v}_{t_n}^{\text{PCA}} \right\}$$

where $\mathbf{v}_{t_i}^{\text{PCA}}$ represents the PCA-reduced embedding vector for the test sample t_i. $\mathcal{T}_{\text{emb}}$ represents the set of PCA-reduced embedding vectors for all test samples $t_1, t_2, \ldots, t_n$.

❷ Next, based on the contrastive learning model trained in Step 3 (cf. Section 6.3.4), LongTest inputs the reduced-dimensional vector of each test t_i into the contrastive learning model to obtain its contrastive embedding vector. This is represented as:

$$\mathcal{T}_{\text{cont}} = \left\{ \mathbf{v}_{t_1}^{\text{cont}}, \mathbf{v}_{t_2}^{\text{cont}}, \ldots, \mathbf{v}_{t_n}^{\text{cont}} \right\}$$

where $\mathbf{v}_{t_i}^{\text{cont}}$ denotes the contrastive embedding vector for each test sample t_i. $\mathcal{T}_{\text{cont}}$ represents the set of contrastive embedding vectors for all test samples

$t_1, t_2, \ldots, t_n$.

❸ Subsequently, based on each sample's contrastive learning vector $\mathbf{v}_{t_i}^{\text{cont}}$, LongTest utilizes the Random Forest model trained in Step 4 (cf. Section 6.3.5) to estimate the probability of each test sample t_i being misclassified by the model M. This is represented as:

$$\mathcal{P}_{\text{mis}} = \{P_{\text{mis}}(t_1), P_{\text{mis}}(t_2), \ldots, P_{\text{mis}}(t_n)\}$$

where $P_{\text{mis}}(t_i)$ represents the probability of test sample t_i being misclassified by the model M. $\mathcal{P}_{\text{mis}}$, represents the set of misclassification probabilities for all test samples $t_1, t_2, \ldots, t_n$.

❹ Finally, LongTest ranks all the tests in the test set T based on their probabilities to be misclassified by the model, resulting in a sorted test set $\mathcal{T}_{\text{ranked}}$:

$$\mathcal{T}_{\text{ranked}} = \{t_{(1)}, t_{(2)}, \ldots, t_{(n)}\}$$

such that:

$$P_{\text{mis}}(t_{(1)}) \geq P_{\text{mis}}(t_{(2)}) \geq \cdots \geq P_{\text{mis}}(t_{(n)})$$

where $P_{\text{mis}}(t_{(i)})$ denotes the probability that the sample $t_{(i)}$ will be misclassified by the model M. The subscript indices $(1), (2), \ldots, (n)$ represent the ranked order of the test samples after sorting by their misclassification probabilities. For example, $t_{(1)}$ refers to the test sample with the highest misclassification probability.

6.4 Study design

In this section, we comprehensively present our study design from various perspectives. Specifically, Section 6.4.1 presents the research questions that served as the guiding framework for our investigation. Section 6.4.2 presents the datasets and models employed to assess the effectiveness of LongTest. Section 6.4.3 presents the measurements we used for evaluation. Section 6.4.4 presents the test prioritization approaches we utilized to compare with LongTest. Section 6.4.5 describes the implementation and configuration setup utilized in our study.

6.4.1 Research Questions

Our experimental evaluation addresses the following research questions.

- **RQ1: How does LongTest perform in prioritizing tests for long text files?**
  We evaluate the performance of LongTest by comparing it with several existing test prioritization approaches [7, 3]. We conduct statistical analysis [233, 234] to prove that the observed improvements achieved by LongTest over the existing approaches are statistically significant.
- **RQ2: How does the number of chunks affect LongTest effectiveness?**
  To perform test prioritization, LongTest first divides a long text file into smaller chunks, converts these chunks into embeddings, and then concatenates them. In this research question, we investigate how splitting into different numbers of chunks impacts the effectiveness of LongTest.
- **RQ3: What is the impact of different embedding models on LongTest?**
  In the test prioritization process of LongTest, chunks of text are converted into vectors using an embedding model. This research question investigates the impact of different embedding models on the effectiveness of LongTest.

- **RQ4: What is the impact of reducing to different dimensions on LongTest?**
  Within the LongTest framework, we apply Principal Component Analysis (PCA) [224] to reduce the dimensionality of text embedding vectors. This research question investigates the impact of reducing text vectors to different dimensions on the effectiveness of LongTest.
- **RQ5: How do the main parameters influence the effectiveness of LongTest?**
  We investigate the impact of main parameters in LongTest to examine its stability across different parameter selections.
- **RQ6: Does each component contribute to LongTest?**
  LongTest comprises core components, including contrastive learning and PCA for dimensionality reduction. This research question investigates the contributions of these components to the effectiveness of LongTest through an ablation study.

6.4.2 Models and Datasets

To evaluate the effectiveness of LongTest and the compared test prioritization approaches, we use a set of 45 subjects, including 3 long-text datasets and 15 text classification models. The essential details and the matching relationship between the long-text datasets and the models are presented in Table 6.1.

6.4.2.1 Datasets

In our research, we utilized three long text datasets: EURLEX57K [235], FakeNews [236], and Cancer Text Documents [222]. The reasons for selecting these datasets to evaluate LongTest are as follows: these datasets have been extensively studied in academia and are widely used and discussed by researchers and developers on data science competition platforms (such as Kaggle). Additionally, these datasets contain a considerable proportion of long texts. For example, the CancerDoc dataset comprises 98.7% long texts, while the EURLEX57K and FakeNews datasets include 51.3% and 32.4% long texts, respectively. Therefore, these datasets are suitable for testing the long text-oriented test prioritization approach LongTest. Here, based on prior research, texts with more than 512 tokens are regarded as long texts [237].

- **EURLEX57K** [235] The EURLEX57K dataset is a large-scale, publicly available dataset consisting of 57k English-language EU legislative documents from the EUR-LEX portal.
- **FakeNews** [236] The FakeNews dataset is a publicly available dataset containing news articles with titles and full-text content. This dataset is designed for text classification tasks aimed at distinguishing between real and fake news.
- **CancerDoc** [222] The Cancer Text Documents dataset is a publicly available dataset consisting of cancer-related research articles, including abstracts and full papers. This dataset focuses on lengthy research papers with more than six pages. It is designed for text classification tasks aimed at analyzing and categorizing cancer-focused content. Examples of labels include "Thyroid_Cancer" and "Colon_Cancer."

6.4.2.2 Models

To evaluate the effectiveness of LongTest, we employed 15 well-established text

Table 6.1: Datasets and Models

ID	Dataset	Ratio of Long Docs	# Size	Models
1	CancerDoc	98.7%	7569	DistilRoBERTa-DNN
2	CancerDoc	98.7%	7569	DistilRoBERTa-DT
3	CancerDoc	98.7%	7569	DistilRoBERTa-LR
4	CancerDoc	98.7%	7569	DistilRoBERTa-RF
5	CancerDoc	98.7%	7569	DistilRoBERTa-TabNet
6	CancerDoc	98.7%	7569	MPNet-DNN
7	CancerDoc	98.7%	7569	MPNet-DT
8	CancerDoc	98.7%	7569	MPNet-LR
9	CancerDoc	98.7%	7569	MPNet-RF
10	CancerDoc	98.7%	7569	MPNet-TabNet
11	CancerDoc	98.7%	7569	MiniLM-DNN
12	CancerDoc	98.7%	7569	MiniLM-DT
13	CancerDoc	98.7%	7569	MiniLM-LR
14	CancerDoc	98.7%	7569	MiniLM-RF
15	CancerDoc	98.7%	7569	MiniLM-TabNet
16	EURLEX57K	51.3%	57000	DistilRoBERTa-DNN
17	EURLEX57K	51.3%	57000	DistilRoBERTa-DT
18	EURLEX57K	51.3%	57000	DistilRoBERTa-LR
19	EURLEX57K	51.3%	57000	DistilRoBERTa-RF
20	EURLEX57K	51.3%	57000	DistilRoBERTa-TabNet
21	EURLEX57K	51.3%	57000	MPNet-DNN
22	EURLEX57K	51.3%	57000	MPNet-DT
23	EURLEX57K	51.3%	57000	MPNet-LR
24	EURLEX57K	51.3%	57000	MPNet-RF
25	EURLEX57K	51.3%	57000	MPNet-TabNet
26	EURLEX57K	51.3%	57000	MiniLM-DNN
27	EURLEX57K	51.3%	57000	MiniLM-DT
28	EURLEX57K	51.3%	57000	MiniLM-LR
29	EURLEX57K	51.3%	57000	MiniLM-RF
30	EURLEX57K	51.3%	57000	MiniLM-TabNet
31	FakeNews	32.4%	44898	DistilRoBERTa-DNN
32	FakeNews	32.4%	44898	DistilRoBERTa-DT
33	FakeNews	32.4%	44898	DistilRoBERTa-LR
34	FakeNews	32.4%	44898	DistilRoBERTa-RF
35	FakeNews	32.4%	44898	DistilRoBERTa-TabNet
36	FakeNews	32.4%	44898	MPNet-DNN
37	FakeNews	32.4%	44898	MPNet-DT
38	FakeNews	32.4%	44898	MPNet-LR
39	FakeNews	32.4%	44898	MPNet-RF
40	FakeNews	32.4%	44898	MPNet-TabNet
41	FakeNews	32.4%	44898	MiniLM-DNN
42	FakeNews	32.4%	44898	MiniLM-DT
43	FakeNews	32.4%	44898	MiniLM-LR
44	FakeNews	32.4%	44898	MiniLM-RF
45	FakeNews	32.4%	44898	MiniLM-TabNet

classification models, specifically: DistilRoBERTa-DT [238, 197], DistilRoBERTa-DT [238, 239], DistilRoBERTa-LR [238, 240], DistilRoBERTa-RF [238, 241], DistilRoBERTa-TabNet [238, 242], MPNet-DNN [243, 197], MPNet-DT [243, 239], MPNet-LR [243, 240], MPNet-RF [243, 241], MPNet-TabNet [243, 242], MiniLM-DNN [230, 197], MiniLM-DT [230, 239], MiniLM-LR [230, 240], MiniLM-RF [230, 241], and MiniLM-TabNet [230, 242]. These models are widely recognized combo models combining embedding and classification components. We selected these models because they are widely utilized in both industry and academia.

Although our evaluation was conducted using these 15 models, the proposed LongTest framework is not limited to these specific models. LongTest is primarily designed to address long-text scenarios and can be applied to any text classification model. This is explained in detail in Section 6.6.1.

6.4.3 Measurements

To evaluate the effectiveness of LongTest and the comparative test prioritization methods, following prior work [3], we used two classic test prioritization metrics: Average Percentage of Fault-Detection (APFD) and Percentage of Fault Detected (PFD). Below, we provide a detailed explanation of these two metrics.

6.4.3.1 Average Percentage of Fault-Detection (APFD)

APFD [92] is a widely used metric for evaluating the effectiveness of test prioritization techniques. Higher APFD values indicate faster rates of detecting misclassified test inputs. The APFD value can be computed using Formula 6.6.

$$\text{APFD} = 1 - \frac{\sum_{i=1}^{M} o_i}{M \cdot N} + \frac{1}{2N} \tag{6.6}$$

where N refers to the total number of test inputs. M denotes the number of misclassified test cases by the model, and o_i refers to the position of the i-th misclassified test case in the test set after prioritization. Following the existing study [3], we normalize the APFD value to $[0, 1]$. If the APFD value of a test prioritization approach is closer to 1, we consider this prioritization method more effective. In the following, we explain the corresponding reasons:

First, the larger the APFD value of a test prioritization method, the smaller the value of $\sum_{i=1}^{M} o_i$ in Formula 6.6, since both M and N are constants. Here, $\sum_{i=1}^{M} o_i$ represents the sum of indices of all misclassified tests within the prioritized test set. A smaller value of this sum indicates that more misclassified tests have been prioritized towards the front, suggesting that the test prioritization method is more effective at detecting misclassifications. Therefore, a test prioritization approach with high APFD is considered more effective.

6.4.3.2 Percentage of Fault Detected (PFD)

PFD calculates the percentage of misclassified tests detected out of all misclassified tests when prioritizing a certain percentage of the data. If a test prioritization method has high PFD values, it means the method can detect more misclassifications and is, therefore, more effective. PFD is calculated based on Formula 7.6. In our experiment, we calculated the PFD values for each test prioritization method when prioritizing

10%, 20%, 30%, 40%, and 50% of the tests, represented as PFD-10, PFD-20, PFD-30, PFD-40, and PFD-50, respectively.

$$\text{PFD} = \frac{|\mathcal{N}_d|}{|\mathcal{N}|} \tag{6.7}$$

where $\mathcal{N}_d$ denotes the set of misclassified tests that have been detected. $\mathcal{N}$ represents the total set of misclassified tests.

6.4.4 Compared Approaches

To evaluate the effectiveness of LongTest, we adopted five test prioritization methods as comparison approaches, including four confidence-based methods and a baseline method (random selection). These prioritization methods were selected for two main reasons: first, all methods can be applied to prioritizing long text files, and their implementations are open-source. Second, the effectiveness of these methods has been demonstrated [7, 3]. Below, we provide a detailed description of the principles and workflows of these prioritization methods.

- **DeepGini** [3] DeepGini is a confidence-based test prioritization approach, which calculates the Gini score of the test to estimate how likely this test will be misclassified. If the score is higher, it means that this test is more likely to be misclassified. The calculation process of the Gini score is presented in Formula 6.8.

$$\text{Gini}(t) = 1 - \sum_{i=1}^{N} \left(p_i(t)\right)^2 \tag{6.8}$$

  where $p_i(t)$ represents the probability that the input t to be classified to category i. N denotes the total number of possible categories.
- **Vanilla Softmax** [7] The Vanilla Softmax (VanillaSM) approach quantifies the uncertainty of the model's prediction for a given test by measuring the difference between the model's most confident prediction and the ideal value of 1. The greater this difference, the less confident the model is in its prediction for this test, indicating that the test is more likely to be misclassified. This difference value is calculated by Formula 6.9.

$$\text{VanillaSM}(t) = 1 - \max_{i=1}^{N} p_i(t) \tag{6.9}$$

  where $p_i(t)$ refers to the probability that the input t is classified into category i. Therefore, $\max_{i=1}^{N} p_i(t)$ denotes the model's highest confidence prediction for the test input t.
- **Prediction-Confidence Score** The Prediction-Confidence Score (PCS) quantifies the model's confidence in its prediction for a given test input by calculating the difference between the probability of the predicted class and that of the second most confident class in the softmax output. If a test's PCS value is small, it indicates that the model is less confident on this test, and this test is more likely to be misclassified. The computation of PCS is presented in Formula 6.10.

$$\text{PCS}(t) = p_{\max}(t) - p_{\text{second}}\,(t) \tag{6.10}$$

  where $p_{\max}(t)$ denotes the probability associated with the model's most confident prediction for the test input t, and $p_{\text{second}}(t)$ represents the probability associated with the model's second most confident prediction for the test input t.

- **Entropy** [7] Entropy measures the model's confidence in a test input by calculating the entropy of the softmax likelihood distribution. If the entropy value for a test input is higher, it indicates that the model is less confident in its prediction, and this test input is more likely to be misclassified.
- **Random selection** [133] Random selection determines the execution order of test inputs in a fully randomized manner.

6.4.5 Implementation and Configuration

We implemented LongTest in Python 3.7.2, utilizing TensorFlow 2.2.0 [244], PyTorch 1.11.0 [203], sklearn 0.24.2 [245], and SentenceTransformer 2.2.2 [246]. For the approaches used for comparison, we employed the available implementations provided by their respective authors [3, 7]. The accuracy of the models used to evaluate LongTest and the comparison approaches is as follows: On the EURLEX57K dataset, the models' accuracy range is 78.5%~86.7%. On the FakeNews dataset, the models' accuracy range is 71.2%~87.5%. On the CancerDoc dataset, the models' accuracy range is 71.4%~92.4%. Our experimental setup involved conducting experiments on NVIDIA Tesla V100 32GB GPUs. For the data analysis, we utilized a MacBook Pro laptop running Mac OS Sonoma 14.3, equipped with an Intel Core i9 CPU and 64 GB of RAM.

6.5 Results and analysis

6.5.1 RQ1: Performance of LongTest

Objectives: We aim to evaluate the effectiveness of LongTest, a novel approach for prioritizing long text files, by comparing its performance with existing test prioritization techniques.

Experimental design: To comprehensively evaluate the performance of LongTest, our experimental design focused on three key aspects: effectiveness evaluation, statistical analysis, and efficiency evaluation.

- **Effectiveness evaluation** To assess the effectiveness of LongTest, we conducted experiments on a total of 45 subjects. Each subject represents a unique model-dataset pair. For instance, the subject (CancerDoc, MiniLM-DT) refers to the case where we use the dataset CancerDoc with the model MiniLM-DT to evaluate the test prioritization approaches. Table 6.1 provides basic information of all the subjects. We selected a set of well-established test prioritization methods (cf. Section 6.4.4) as the comparison approaches and adopted two classical evaluation metrics, namely APFD and PFD, for evaluation. Detailed explanations of these metrics can be found in Section 6.4.3.
- **Statistical analysis** Due to the inherent randomness in the model training process, we conducted statistical analysis by running all the experiments ten times and reported the average results. Moreover, we calculated the p-value and effect size to validate the stability and statistical significance of the experimental results.
  - **P-value:** To calculate the p-value, we employed the **paired two-sample t-test** [233], a widely used statistical method for evaluating differences between two data sets. Based on prior work [46], we consider differences between two data sets to be statistically significant if the p-value is less than 10^{-5}.
  - **Effect size:** We quantified the magnitude of the difference between the two sets of results using effect size, specifically employing Cohen's d [234] as a

measure. Here, $|d| < 0.2$ implies a "negligible" effect, $|d| < 0.5$ implies a "small" effect, $|d| < 0.8$ implies a "medium" effect, and values above 0.8 imply a "large" effect. To ensure that the difference between the results of LongTest and the compared approach is "non-negligible", we require that the value of $d \geq 0.2$.

- **Efficiency evaluation** We evaluated the efficiency of LongTest and the compared test prioritization approaches by quantifying their runtime. Our goal is to gain insights into the computational efficiency and potential for practical application of LongTest.

Results: The experimental results of RQ1 is presented in Table 6.2, Table 6.3, Table 6.4, Table 6.5 and Table 6.6. These tables respectively show the effectiveness evaluation using APFD, the statistical analysis, the effectiveness evaluation using PFD, and the efficiency evaluation.

Effectiveness comparison in terms of APFD Table 6.2 presents the effectiveness comparison between LongTest and the other approaches across different datasets and models using the APFD metric. Gray highlights indicate the best test prioritization method for a particular case. In Table 6.2, we see that LongTest outperforms all the compared approaches in each case. The APFD of LongTest ranges from 0.710 to 0.980, while the APFD values for the compared approaches range from 0.484 to 0.846. Table 6.3 presents an overall comparison between LongTest and the compared approaches, including the best cases achieved by each approach, the average APFD of each approach, and the average improvement of LongTest over the other test prioritization methods. We see that the average effectiveness of LongTest is 0.856, while the average APFD values of the compared test prioritization methods range from 0.501 to 0.749. The improvement of LongTest, relative to the comparative methods, ranges from 14.28% to 70.86%. In conclusion, based on the APFD metric, we find that LongTest outperforms all the compared test prioritization approaches.

Statistical analysis To demonstrate the statistical significance and stability of the experimental results, we conducted a statistical analysis, and the experimental results are presented in Table 6.4. Following the existing study [233], if the p-value between LongTest and a comparative method is less than 10^{-5}, we consider that the improvement achieved by LongTest has statistical significance. In Table 6.4, we see that the p-values between LongTest and each test prioritization method are all less than 10^{-5} (ranging from 6.302×10^{-17} to 1.249×10^{-09}). These experimental results indicate that LongTest outperforms all the test prioritization approaches with statistical significance.

Effectiveness comparison in terms of PFD Table 6.5 presents the average comparison results between LongTest and the compared approaches in terms of PFD. We can see that LongTest demonstrates the highest effectiveness across all cases when prioritizing tests at different proportions. Notably, when prioritizing the top 50% of tests, LongTest identifies between 92.2% and 99.8% of misclassified tests. Based on these experimental results, we conclude that, in terms of the PFD metric, LongTest outperforms all other test prioritization approaches.

Efficiency evaluation Table 6.6 presents the efficiency evaluation of LongTest and the compared test prioritization methods by assessing their running time. From the table, we see that the total running time of LongTest is less than 8 minutes, with *text embedding* taking 5 minutes, *contrastive learning* taking 2 minutes, and *prediction* taking less than 1 second. Although LongTest is not as efficient as confidence-based test prioritization approaches, it achieves an average effectiveness improvement of

Table 6.2: Effectiveness comparison among LongTest, DeepGini, VanillaSM, PCS, Entropy and random selection in terms of the APFD values

Data	Model	Approach					
		Random	DeepGini	VanillaSM	PCS	Entropy	**LongTest**
CancerDoc	DistilRoBERTa-DNN	0.511	0.650	0.693	0.680	0.647	0.861
	DistilRoBERTa-DT	0.518	0.753	0.754	0.753	0.753	0.930
	DistilRoBERTa-LR	0.511	0.828	0.837	0.845	0.814	0.980
	DistilRoBERTa-RF	0.525	0.734	0.760	0.769	0.723	0.912
	DistilRoBERTa-TabNet	0.495	0.735	0.734	0.733	0.738	0.893
	MPNet-DNN	0.512	0.638	0.662	0.676	0.620	0.865
	MPNet-DT	0.515	0.689	0.693	0.689	0.691	0.875
	MPNet-LR	0.488	0.695	0.701	0.702	0.685	0.879
	MPNet-RF	0.515	0.704	0.725	0.732	0.693	0.889
	MPNet-TabNet	0.511	0.740	0.741	0.741	0.740	0.908
	MiniLM-DNN	0.492	0.641	0.646	0.646	0.630	0.865
	MiniLM-DT	0.506	0.702	0.704	0.704	0.701	0.858
	MiniLM-LR	0.490	0.686	0.686	0.683	0.682	0.894
	MiniLM-RF	0.487	0.678	0.701	0.710	0.673	0.870
	MiniLM-TabNet	0.522	0.704	0.703	0.702	0.705	0.865
EURLEX57K	DistilRoBERTa-DNN	0.494	0.806	0.806	0.806	0.806	0.900
	DistilRoBERTa-DT	0.505	0.755	0.755	0.754	0.754	0.808
	DistilRoBERTa-LR	0.500	0.756	0.755	0.755	0.755	0.883
	DistilRoBERTa-RF	0.497	0.797	0.799	0.798	0.793	0.881
	DistilRoBERTa-TabNet	0.522	0.835	0.836	0.837	0.835	0.893
	MPNet-DNN	0.488	0.829	0.823	0.819	0.838	0.908
	MPNet-DT	0.497	0.767	0.768	0.768	0.766	0.825
	MPNet-LR	0.503	0.789	0.790	0.790	0.785	0.895
	MPNet-RF	0.491	0.821	0.821	0.818	0.817	0.877
	MPNet-TabNet	0.499	0.817	0.817	0.817	0.818	0.868
	MiniLM-DNN	0.491	0.845	0.843	0.842	0.846	0.918
	MiniLM-DT	0.496	0.743	0.743	0.743	0.741	0.814
	MiniLM-LR	0.493	0.836	0.834	0.834	0.838	0.911
	MiniLM-RF	0.488	0.796	0.796	0.795	0.791	0.880
	MiniLM-TabNet	0.493	0.776	0.780	0.783	0.761	0.895
FakeNews	DistilRoBERTa-DNN	0.497	0.737	0.746	0.753	0.720	0.868
	DistilRoBERTa-DT	0.484	0.697	0.697	0.695	0.695	0.734
	DistilRoBERTa-LR	0.504	0.697	0.728	0.737	0.684	0.844
	DistilRoBERTa-RF	0.501	0.686	0.700	0.708	0.676	0.790
	DistilRoBERTa-TabNet	0.504	0.776	0.768	0.758	0.788	0.836
	MPNet-DNN	0.496	0.747	0.759	0.769	0.735	0.842
	MPNet-DT	0.499	0.693	0.694	0.691	0.693	0.720
	MPNet-LR	0.497	0.730	0.759	0.764	0.717	0.809
	MPNet-RF	0.502	0.700	0.711	0.714	0.693	0.783
	MPNet-TabNet	0.498	0.780	0.779	0.778	0.781	0.835
	MiniLM-DNN	0.495	0.689	0.705	0.711	0.680	0.849
	MiniLM-DT	0.499	0.645	0.645	0.644	0.644	0.710
	MiniLM-LR	0.494	0.727	0.738	0.741	0.716	0.804
	MiniLM-RF	0.512	0.704	0.723	0.737	0.692	0.817
	MiniLM-TabNet	0.503	0.750	0.752	0.754	0.746	0.794

Table 6.3: Effectiveness improvement of LongTest over the compared approaches in terms of the APFD values

Approach	# Best cases	Average APFD	Improvement(%)
Random	0	0.501	70.86
DeepGini	0	0.741	15.52
VanillaSM	0	0.747	14.59
PCS	0	0.749	14.28
Entropy	0	0.736	16.31
LongTest	45	0.856	-

Table 6.4: Statistical analysis on test inputs (in terms of p-value and effect size)

	Random	DeepGini	VanillaSM	PCS	Entropy
LongTest (p-value)	6.302×10^{-17}	1.632×10^{-09}	1.249×10^{-09}	2.085×10^{-09}	1.267×10^{-09}
LongTest (effect size)	6.273	2.403	2.442	2.367	2.441

Table 6.5: Average comparison results among LongTest and the compared approaches in terms of PFD

Data	Approach	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50
CancerDoc	Random	0.104	0.204	0.304	0.402	0.496
	DeepGini	0.237	0.423	0.580	0.705	0.798
	VanillaSM	0.253	0.439	0.601	0.728	0.815
	PCS	0.253	0.446	0.599	0.728	0.822
	Entropy	0.235	0.424	0.571	0.695	0.790
	LongTest	0.481	0.854	0.997	0.998	0.998
EURLEX57K	Random	0.097	0.196	0.297	0.399	0.501
	DeepGini	0.331	0.584	0.776	0.886	0.940
	VanillaSM	0.329	0.585	0.775	0.885	0.939
	PCS	0.328	0.585	0.775	0.884	0.939
	Entropy	0.325	0.575	0.773	0.885	0.940
	LongTest	0.537	0.835	0.933	0.967	0.981
FakeNews	Random	0.096	0.199	0.299	0.398	0.499
	DeepGini	0.246	0.440	0.601	0.731	0.825
	VanillaSM	0.257	0.465	0.626	0.748	0.835
	PCS	0.259	0.471	0.633	0.755	0.843
	Entropy	0.244	0.430	0.588	0.713	0.812
	LongTest	0.401	0.631	0.775	0.864	0.922

14.28%~70.86% compared to confidence-based methods. Considering the trade-off between effectiveness and efficiency, LongTest remains a practical option.

Answer to RQ1: *LongTest outperforms all the compared test prioritization approaches (i.e., DeepGini, Vanilla SM, PCS, Entropy, and Random), with an average improvement of 14.28%~70.86%.*

6.5.2 RQ2: Impact of Number of Chunks on LongTest

Objectives: In the LongTest workflow, an input test (a long text file) is first divided into multiple small chunks. The reason for splitting the long text into chunks is that the embedding models used to convert long text into embeddings typically have input token limitations, meaning they are constrained by the length of the input text and cannot process information beyond this limit. By dividing the long text into smaller chunks, we aim to better capture information from the entire text.

However, the number of chunks can influence the quality of the generated em-

Table 6.6: Time cost of LongTest and the compared test prioritization approaches

Time cost	Approach					
	LongTest	Random	DeepGini	VanillaSM	PCS	Entropy
Text Embedding	5 min	-	-	-	-	-
Contrastive Learning	2 min	-	-	-	-	-
Prediction	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s

Table 6.7: The PFD values of LongTest with different numbers of chunks

Chunks	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50
Chunk-5	0.372	0.592	0.738	0.838	0.905
Chunk-10	0.401	0.631	0.775	0.864	0.923
Chunk-15	0.405	0.641	0.788	0.875	0.930
Chunk-20	0.412	0.652	0.793	0.882	0.934

beddings, thereby affecting the overall effectiveness of LongTest. For instance, if the number of chunks is too small, the chunk size can exceed the input limitations of the embedding model, resulting in information loss. In this research question, we explored the impact of the number of chunks on the effectiveness of LongTest.

Experimental design: To investigate the impact of chunk number on the effectiveness of LongTest, we kept all other workflows in LongTest unchanged, adjusting only the chunk number to values of 5, 10, 15, and 20, and compared LongTest's effectiveness. We used the metrics APFD and PFD for evaluation.

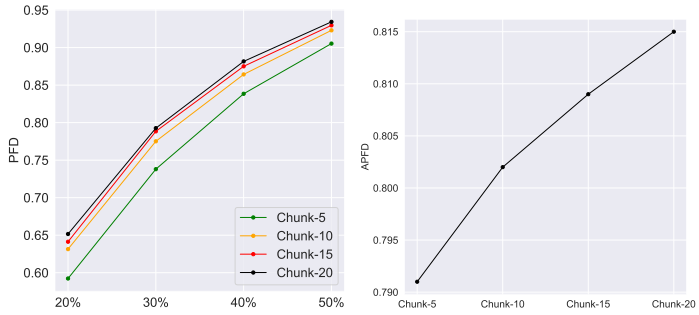
Results: The experimental results for RQ2 are presented in Table 6.7 and Figure 6.2. Table 6.7 shows the effectiveness of LongTest when the input long text file is divided into different numbers of chunks. We see that as the number of chunks increases, the effectiveness of LongTest improves. Figure 6.2 visually demonstrates this relationship. Specifically, Figure 6.2a) shows the evaluation using the PFD metric, where the X-axis represents the percentage of tests executed, and the Y-axis represents the effectiveness of LongTest. We see that LongTest with 20 chunks (represented by a black line) achieves the highest effectiveness. As the number of chunks increases, the effectiveness of LongTest gradually rises. Figure 6.2b) presents the evaluation results based on the APFD metric, with the X-axis representing the number of chunks and the Y-axis representing the APFD value of LongTest. We also see that as the number of chunks increases, the effectiveness of LongTest improves.

However, although increasing the number of chunks can improve the effectiveness of LongTest, the improvement effect gradually slows down. From Figure 6.2a), we see that the gap between the curves for Chunk-10 and Chunk-15 is much smaller than the gap between Chunk-5 and Chunk-10, and the curve for Chunk-20 nearly overlaps with that of Chunk-15. This indicates that as the number of chunks continues to increase, the growth in effectiveness becomes limited.

> **Answer to RQ2:** *When the number of chunks increases, the effectiveness of LongTest improves. However, as the number of chunks continues to increase, the growth in effectiveness becomes limited.*

6.5.3 RQ3: Impact of Different Embedding Models on LongTest

Objectives: Within the LongTest framework, text chunks are converted into embedding vectors, which serve as the foundation for test prioritization. Different embedding models can capture different semantic nuances, contextual relationships, and feature representations, impacting the effectiveness of LongTest. By investigating



a) PFD **b)** APFD

Figure 6.2: The APFD and PFD values of LongTest with different numbers of chunks

Table 6.8: The APFD values of LongTest with different embedding models

Embedding Model	Dataset			Average
	CancerDoc	EURLEX57K	FakeNews	
DistilRoBERTa	0.916	0.873	0.815	0.868
MPNet	0.883	0.875	0.798	0.852
MiniLM	0.871	0.883	0.795	0.849

the effectiveness of Longtest with different embedding models, we aim to identify the models that are more suitable for LongTest to perform test prioritization.

Experimental design: To investigate the impact of different embedding models on LongTest, we studied three popular embedding models: DistilRoBERTa [238], MPNet [243], and MiniLM [230]. The core function of these models is to convert text into embeddings, which involves transforming the input text into vector representations to capture semantic information. **1) DistilRoBERTa** is a simplified version of the RoBERTa model [238]. RoBERTa is a pre-trained language model particularly skilled in capturing subtle semantics and contextual relationships within text. DistilRoBERTa achieves higher computational efficiency through distillation, maintaining strong semantic understanding capabilities while reducing resource demands. **2) MPNet** combines the advantages of BERT [247] and XLNet [248], utilizing parallel masking and positional encoding to improve the model's performance in contextual semantic understanding. **3) MiniLM** is a lightweight transformer model that leverages deep distillation techniques to strike a balance between efficient performance and embedding quality.

Specifically, in our study, we kept other workflows within LongTest unchanged, modifying only the embedding models used. We then evaluated LongTest's effectiveness with each embedding model using APFD and PFD metrics.

Results: The experimental results for RQ3 are presented in Table 6.8 and Table 6.9. Table 6.8 shows the effectiveness of LongTest when using different embedding models, evaluated by APFD. Specifically, with the embedding model DistilRoBERTa, LongTest achieves the highest average effectiveness (0.868), surpassing the results with embedding models MPNet (0.852) and MiniLM (0.849). Table 6.9 presents the evaluation results with PFD as the metric. We see that, regardless of the ratio of test inputs prioritized, LongTest's effectiveness is consistently the best when using DistilRoBERTa as the embedding model. The above experimental results demonstrate that, compared to the embedding model MPNet and MiniLM, LongTest with DistilRoBERTa performs better in most cases.

Table 6.9: The PFD values of LongTest with different embedding models

Embedding Model	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50
DistilRoBERTa	0.511	0.802	0.909	0.949	0.972
MPNet	0.452	0.769	0.896	0.937	0.963
MiniLM	0.454	0.749	0.899	0.942	0.967

Table 6.10: The APFD values of LongTest with different dimensions

Dimension Vector	Dataset			Average
	CancerDoc	EURLEX57K	FakeNews	
32	0.870	0.868	0.765	0.834
64	0.871	0.881	0.787	0.846
128	0.890	0.877	0.802	0.856
256	0.864	0.878	0.798	0.847

Answer to RQ3: *LongTest with the embedding model DistilRoBERTa performs better in most cases compared to MPNet and MiniLM.*

6.5.4 RQ4: Impact of Dimension Reduction on LongTest

Objectives: Within the LongTest framework, a crucial step is using the PCA algorithm [224] for dimensionality reduction, which involves reducing the original high-dimensional embedding vectors of long text to low-dimensional vectors. This approach aims to reduce the overall runtime and enhance the efficiency of test prioritization while preserving the features of the original data. In this research question, we investigate the impact of reducing to different dimensions on the effectiveness of LongTest. This analysis helps identify the optimal dimensionality reduction level that preserves semantic information while improving computational efficiency.

Experimental design: To investigate the impact of different dimensions on LongTest, we selected the dimensions 32, 64, 128, and 256, which are common dimensions in the literature [249]. We used APFD and PFD as evaluation metrics to assess the effectiveness of LongTest with these different dimensions.

Throughout the experiment, we kept all other workflows in LongTest unchanged, adjusting only the dimensionality reduced by PCA to study its effect on LongTest. We repeated the experiment across different datasets and models to validate the stability and generalizability of the results.

Results: The experimental results of RQ4 are presented in Table 6.10 and Table 6.11. Table 6.10 presents the effectiveness of LongTest with different dimensions measured by APFD. We see that LongTest with a dimension of 128 achieves the highest effectiveness (with an APFD of 0.856), while LongTest with dimensions of 256, 64, and 32 results in effectiveness values of 0.847, 0.846, and 0.834, respectively. Table 6.9 provides further evaluations using PFD. In Table 6.9, we see that at various test prioritization ratios, LongTest with a dimension of 128 consistently achieves the highest average effectiveness. These results indicate that a 128-dimensional reduction optimally preserves LongTest's effectiveness.

Answer to RQ4: *Within the LongTest framework, when utilizing the PCA algorithm to reduce the dimensionality of the input long text files, the 128-dimensional reduction optimally preserves LongTest's effectiveness.*

Table 6.11: The PFD values of LongTest with different dimensions

Dimension Vector	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50
32	0.412	0.698	0.862	0.915	0.951
64	0.439	0.729	0.886	0.934	0.963
128	0.473	0.773	0.902	0.943	0.967
256	0.451	0.748	0.893	0.934	0.959

6.5.5 RQ5: Impact of Main Parameters on LongTest

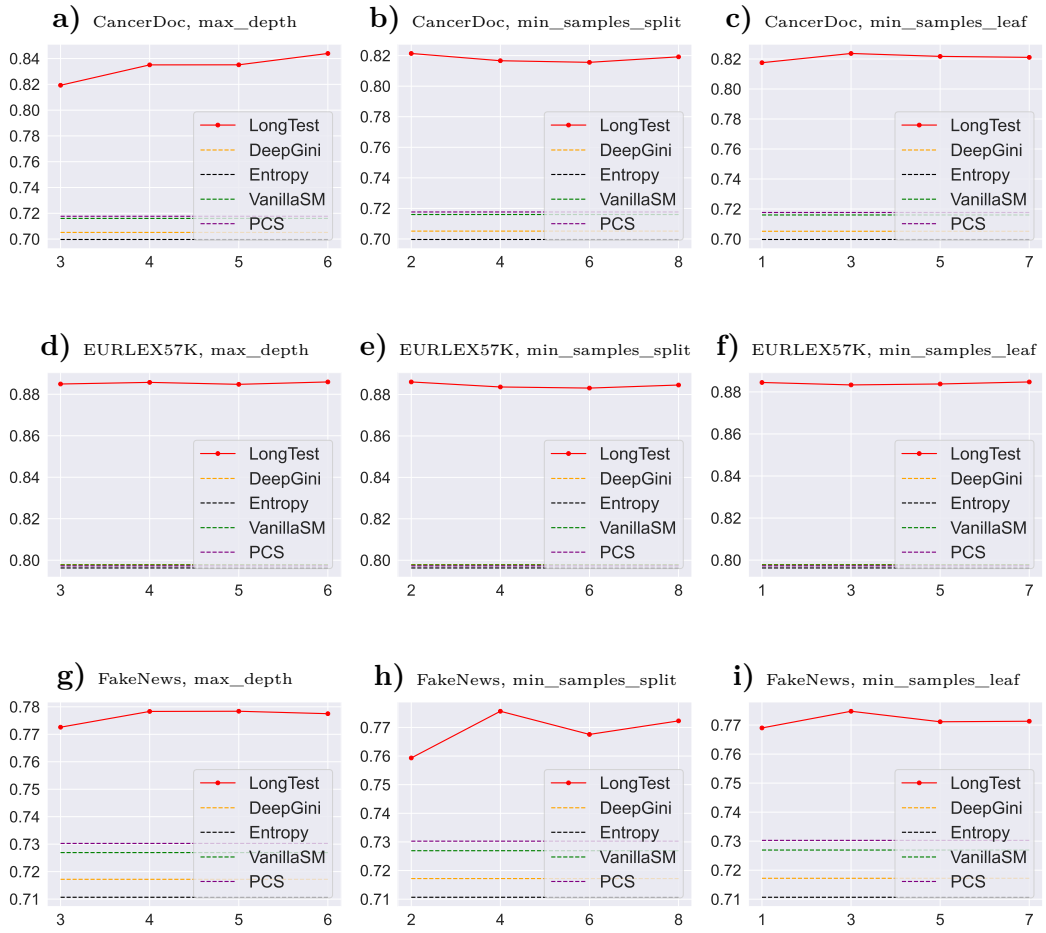


Figure 6.3: Impact of main parameters in LongTest

Objectives: We investigate the impact of the main parameters in LongTest. LongTest uses Random Forest [241] to predict the misclassification probability of a test input. We aim to assess whether LongTest can maintain stability in effectiveness when the main parameters change. The main parameters include: `max_depth` (the maximum depth of each tree in the Random Forest), `min_samples_split` (the minimum number of samples required to split an internal node), and `min_samples_leaf` (the minimum number of samples required to be at a leaf node) in the Random Forest ranking algorithm.

Experimental design: Following existing research [2], we varied each main parameter across a range of values and evaluated the extent to which the effectiveness of LongTest fluctuates. In each evaluation during the experiment, we ensured that the rest of LongTest's workflow remained unchanged, only altering the value of the specific main parameter. We performed experiments on all subjects within the dataset, aiming to study whether LongTest can maintain stability when the values

of these main parameters change.

Results: The experimental results of RQ5 are presented in Figure 6.3, which illustrates the impact of three main parameters `max_depth`, `min_samples_split`, and `min_samples_leaf` on the effectiveness of LongTest across different datasets. Each row represents one of these datasets, and each column represents one parameter being varied, with values shown along the x-axis. The y-axis indicates the test prioritization effectiveness (evaluated by APFD). Here, the solid red line represents LongTest, while the dashed lines present the compared test prioritization approaches.

In Figure 6.3, we see that, across all datasets, regardless of changes in parameter values, LongTest consistently outperforms the compared test prioritization methods (i.e., DeepGini, Entropy, VanillaSM, and PCS). On the CancerDoc dataset, when the values of the main parameters vary, the fluctuation range of LongTest's effectiveness is around 0.01 to 0.02. For example, when the value of the parameter `min_samples_split` changes, LongTest's effectiveness fluctuates between 0.81 and 0.82. On the EURLEX57K dataset, when the range of main parameter values changes, the fluctuation range of LongTest's effectiveness is less than 0.1. For example, when the value of `min_samples_leaf` changes, LongTest remains stable around 0.88 regardless of the parameter change. On the FakeNews dataset, when the main parameter values vary, the fluctuation range of LongTest's effectiveness is around 0.01. The experimental results demonstrate that LongTest performs stably under different parameter settings.

> **Answer to RQ5:** *LongTest consistently outperforms the compared test prioritization methods across various parameter settings.*

6.5.6 RQ6: Contributions of Core Components to LongTest

Objectives: LongTest utilizes some core components to perform test prioritization, including *contrastive learning* and *PCA-based dimensionality reduction*. Specifically, within the LongTest framework, the contrastive learning model can bring the embedding vectors of misclassified tests closer to each other in the space while pushing the misclassified tests farther from the correctly classified ones. The objective is to enable better classification between misclassified and correctly classified samples. PCA-based dimensionality reduction aims to map the original high-dimensional vectors of long text files to a lower-dimensional space, thereby reducing overall runtime and improving the efficiency of test prioritization, while preserving the feature information of the original data. In this research question, we conducted an ablation study to investigate the contributions of each component to LongTest.

Experimental design: We conducted an ablation study to investigate the contributions of each core component to the effectiveness of LongTest. Following the existing approach [250], the specific experimental procedure is as follows:

- We evaluate the test prioritization effectiveness of LongTest when all core components (including contrastive learning and dimensionality reduction) are applied.
- We evaluate the effectiveness of LongTest without applying contrastive learning.
- We evaluate the effectiveness of LongTest without applying both contrastive learning and dimensionality reduction.
- Finally, we compare the changes in the effectiveness of LongTest across these different scenarios.

Results: The experimental results for RQ6 can be found in Table 6.12, which

Table 6.12: Ablation study results

Approach	Dataset			Average
	CancerDoc	EURLEX57K	FakeNews	
LongTest (None)	0.817	0.796	0.672	0.762
LongTest (PCA)	0.811	0.801	0.687	0.766
LongTest (PCA + Contrastive Learning)	0.889	0.877	0.802	0.856

presents the ablation study results to evaluate the impact of different components in LongTest. The study compares three configurations of LongTest:

- **LongTest (None):** the scenario where LongTest does not use either PCA or contrastive learning for test prioritization.
- **LongTest (PCA):** the scenario where LongTest uses dimensionality reduction with PCA but does not use contrastive learning.
- **LongTest (PCA + Contrastive Learning):** the scenario where LongTest uses both dimensionality reduction and contrastive learning for test prioritization.

From Table 6.12, we see that **LongTest (PCA + Contrastive Learning)** has an average effectiveness of 0.856, while **LongTest (PCA)** has an average APFD of 0.766. This case indicates that removing *contrastive learning* leads to an approximate 0.09 reduction in average APFD. These results demonstrate that the *contrastive learning* component contributes to the overall effectiveness of LongTest.

In Table 6.12, we see that the average APFD of **LongTest (PCA)** is 0.766, while the average APFD of **LongTest (None)** is 0.762. This indicates that using *PCA-based dimensionality reduction* slightly improves model performance. The main purpose of applying *PCA-based dimensionality reduction* is to decrease the runtime of LongTest by processing vectors with fewer dimensions, thereby enhancing overall efficiency. As evidenced by prior research [251, 252], the PCA algorithm has proven effective in enhancing the efficiency of data processing in models. Therefore, based on the experimental results, we conclude that the contribution of *dimensionality reduction* to LongTest lies in enabling it to operate more efficiently while maintaining the effectiveness.

Answer to RQ6: *The core component contrastive learning contributes to the effectiveness of LongTest, while the core component dimensionality reduction enables LongTest to operate more efficiently while maintaining the effectiveness.*

6.6 Discussion

In this section, we discuss the generality of LongTest and elaborate on the potential threats to its validity. Specifically, we explain why LongTest can be applied to a wide range of long-text classification tasks and analyze potential internal and external threats that may affect the reliability of our results.

6.6.1 Generality of LongTest

Although our evaluation was conducted using 15 text classification models and three long text datasets (cf. Section 6.4.2), the proposed LongTest framework is not limited to these specific subjects. LongTest is primarily designed to address the test prioritization problem in long-text scenarios and can be applied to any text classification models. This is because the core steps of LongTest include text splitting, embedding generation, embedding concatenation, dimensionality reduction, and contrastive learning. These procedures are applicable to any classification task

with long-text data. By leveraging the features of long texts, LongTest can effectively capture the entire textual information. Through its contrastive learning mechanism, LongTest enhances the ability to distinguish between misclassified and correctly classified samples, thereby improving the effectiveness of test prioritization.

6.6.2 Threats to Validity

Threats to Internal Validity. The internal threats to validity primarily stem from the implementation of our proposed LongTest framework and the test prioritization approaches used for comparison. To mitigate this threat, we implemented LongTest using widely recognized libraries, including PyTorch [203], TensorFlow [244], sklearn [245], and SentenceTransformer [246] (specific versions are detailed in Section 6.4.5). For the compared test prioritization methods, we utilized the original implementations provided by their authors to minimize potential biases introduced during re-implementation. Another internal threat arises from the randomness in the model training process. To mitigate this threat, we conducted a statistical study by repeating all the experiments ten times and reporting the average experimental results. Moreover, we assessed the statistical significance of the results by calculating the p-value [233] and effect size [234]. This approach helps reduce the impact of randomness on our experimental findings.

Threats to External Validity. External threats to validity in our study primarily arise from the long text datasets and text classification models utilized. To mitigate this threat, we employed 15 diverse text classification models, ensuring broad applicability. Additionally, we selected datasets from three different domains: medical (Cancer Text Documents), legal (EURLEX57K), and news articles (FakeNews), to capture a variety of long-text characteristics.

6.7 Related Work

6.7.1 Test Prioritization for Traditional Software

In traditional software testing [169, 253, 140, 254, 133, 139, 255, 92, 256, 257, 258], test prioritization is also an important direction to improve the efficiency of testing, which aims to prioritize all the tests to reveal software bugs as early as possible. In the literature, Jiang *et al.* [137] proposed an adaptive random test case prioritization (TCP) approach that uses test distance to arrange the order of test execution. Thomas *et al.* [253] introduced a novel static black-box TCP method that examines linguistic data within test cases, such as identifier names, comments, and string literals, to represent the test cases. This technique employs topic modeling to process this linguistic information, estimating the specific functionality each test case targets and prioritizing test cases that cover different functionalities. Lou *et al.* [141] developed a mutation-based prioritization approach targeting the sequencing of test cases during software evolution. By generating mutation faults based on code modifications between software versions, this approach simulates realistic faults to improve prioritization relevance. The method employs statistical and probabilistic models to quantify each test case's fault-detection potential based on its effectiveness in "killing" mutants.

Chen *et al.* [259] conducted an empirical evaluation of various TCP techniques and developed a machine learning-based system to recommend the best prioritization method for specific projects by leveraging test distribution patterns. Pan *et al.* [260]

conducted a systematic literature review on the application of machine learning (ML) techniques in test case prioritization. Their findings indicate that ML-based prioritization methods primarily utilize features such as execution history, coverage information, code complexity, and textual data, with supervised learning being the most commonly employed technique. More recently, Chen *et al.* [261] introduced LogTCP, a framework that enhances black-box test case prioritization by analyzing test execution logs to better capture test behaviors.

6.7.2 Testing Deep Learning Systems

Beyond test input prioritization, DNN testing [262, 146, 263, 264, 265, 266] also includes other critical areas, such as accuracy estimation [36, 35, 267] and adequacy measurement [4, 5, 49, 51]. Accuracy estimation aims to improve the efficiency of DNN testing by selecting a representative subset of the original test set to estimate the model's accuracy on the entire dataset. In the literature, Li *et al.* [36] proposed Cross Entropy-based Sampling (CES) for accuracy estimation, which minimizes the cross-entropy between the distribution of the selected subset and the original test set to achieve a representative estimation. Chen *et al.* [35] proposed Practical Accuracy Estimation (PACE), which employs a clustering-based approach. Specifically, PACE first clusters the test inputs into different groups, then utilizes the MMD-critic algorithm [37] to select prototypes from each cluster, thus achieving a practical estimation of accuracy.

For adequacy measurement [4, 5, 49], Pei *et al.*[4] proposed neuron coverage as a metric to evaluate the extent to which a test set activates the internal logic of a DNN model. Expanding on their work, Ma *et al.*[5] proposed DeepGauge, a tool that broadens neuron coverage by introducing a more comprehensive suite of criteria to assess DNN adequacy. Additionally, Kim *et al.* [49] presented the concept of surprise adequacy, which measures test input effectiveness by analyzing the behavioral variations a DL model exhibits between the training and testing set.

6.8 Conclusion

To address the challenge of high labeling costs for long text files, we propose a novel approach called LongTest, which prioritizes test inputs that are more likely to be misclassified. LongTest leverages the unique characteristics of long text files for test prioritization by incorporating two core components: **1) Embedding Generation Mechanism.** This mechanism is specifically designed to enhance the capture of information from the entire long text file. Specifically, for a long text file, LongTest divides it into smaller chunks, converts each chunk into an embedding, and concatenates all chunk embeddings to produce a final embedding vector. LongTest then applies Principal Component Analysis (PCA) to the embedding vector, aiming to decrease its dimensionality while preserving the essential characteristics of the original data. **2) Contrastive Learning.** Contrastive learning brings the embeddings of misclassified samples closer while pushing the embeddings of misclassified and correctly classified samples further apart. This approach enables more effective differentiation between misclassified and correctly classified samples, thereby improving the effectiveness of test prioritization. We conducted an extensive evaluation of LongTest involving 45 subjects, covering three datasets and 15 DNN models. The experimental results demonstrate the effectiveness of LongTest. Specifically, LongtTest outperforms all the compared test prioritization approaches, achieving an average improvement ranging

from 14.28% to 70.86%.

Availability. All artifacts are available in the following public repository:

`https://github.com/yinghuali/LongTest`

7 Towards Exploring the Limitations of Test Selection Techniques on Graph Neural Networks: An Empirical Study

In this chapter, we conduct an empirical study on existing Deep Neural Network (DNN) test selection approaches for Graph Neural Networks (GNNs). Unlike DNNs, GNN test inputs exhibit interdependencies, which may reduce the effectiveness of existing selection methods. In our study, we evaluated 22 test selection approaches across 7 graph datasets and 8 GNN models, focusing on assessing the effectiveness of existing test selection approaches in the context of misclassification detection, accuracy estimation, and performance enhancement, respectively.

This chapter is based on the work published in the following research paper:

- Xueqi Dang, Yinghua Li, Wei Ma, Yuejun Guo, Qiang Hu, Mike Papadakis, Maxime Cordy, Yves Le Traon. Towards Exploring the Limitations of Test Selection Techniques on Graph Neural Networks: An Empirical Study. Empirical Software Engineering (EMSE). Accepted for publication on July. 22, 2024.

Contents

7.1 Introduction

Graph Neural Networks (GNNs) have emerged as powerful tools across a wide range of domains, such as social network analysis [158, 159, 160], recommendation systems [155, 156, 157], and drug discovery [161, 162]. Their ability to capture intricate relationships within graph-structured data has driven significant advancements in the fields of machine learning and artificial intelligence [53]. As the application of GNNs continues to expand, the need for effective testing and evaluation methods becomes increasingly critical.

Similar to traditional deep neural networks (DNNs), testing GNNs faces challenges due to the lack of automated testing oracles [32]. As a result, labeling GNN test inputs heavily relies on manual annotation, which can be expensive and time-consuming, especially when dealing with large and intricate graphs. Furthermore, in specific specialized domains such as molecular property prediction [268], where nodes represent atoms and edges represent covalent bonds, the labeling process can heavily rely on domain-specific knowledge, further increasing the expenses.

In the literature [46, 35, 2], a promising approach for mitigating labeling costs is *test selection*. It focuses on the selection and labeling of a subset of data from the entire test set. Within the field of DNN testing, various test selection techniques have emerged. These techniques can be broadly classified into two categories: 1) test selection for rapid detection of potentially misclassified tests [46, 3] and 2) test selection for precise accuracy estimation [35, 36]. For simplicity, we refer to these approaches as misclassification detection approaches and accuracy estimation approaches, respectively.

Misclassification detection approaches are designed to identify test inputs that are most likely to be misclassified by the DNN model. These selected inputs serve two primary purposes: facilitating the debugging of DNN-based software and retraining the original DNN model to enhance its accuracy [3]. In the literature, there are three main methods for misclassification detection: 1) Coverage-Based Methods [5, 4]: These methods assess the coverage of DNN neurons to identify potentially misclassified test inputs; 2) Surprise Adequacy-Based Methods [49]: These techniques select test inputs using metrics related to surprise adequacy and activation traces within DNNs; 3) Confidence-Based Approaches [3, 46, 7]: These methods selects tests based on the model's prediction confidence. Test inputs that the DNN model is more uncertain are selected. Notably, confidence-based metrics have proven to be more effective and efficient than both surprise adequacy and coverage-based approaches, with runtime typically taking less than 1 second in most cases [3].

Accuracy estimation approaches aim to select a small set of test inputs to precisely estimate the accuracy of the whole testing set. By only labeling the selected representative tests, it becomes feasible to reduce the labeling expenses. However, existing approaches designed for DNNs, like CES [36] and PACE [35], are not suitable for GNNs due to their design not aligning with graph datasets.

GNNs fundamentally belong to the family of DNN algorithms. They inherit several core concepts from DNNs, such as deep architectures, nonlinear activation functions, and backpropagation algorithms. Therefore, several existing DNN test selection approaches can be applied to GNNs. However, there is a significant gap in adapting DNN test selection methods for GNNs. This challenge arises because, unlike

DNNs, where each sample in the test set is treated independently, GNNs exhibit interdependencies among their test inputs (nodes) [269]. Consequently, it remains unclear whether test selection approaches designed for DNNs can be effectively utilized for GNNs. Therefore, it is crucial to investigate the effectiveness of DNN test selection methods in the context of GNNs. To fill the gap, we conduct an empirical study to evaluate the effectiveness of test selection methods when applied to GNNs. Our research focuses on four key aspects:

- **Test Selection for Misclassification Detection** As previously mentioned, confidence-based metrics have demonstrated higher effectiveness and efficiency compared to other existing test selection approaches [3]. Therefore, we specifically evaluate the effectiveness of various confidence-based test selection approaches for selecting potentially misclassified GNN test inputs.
- **Test Selection for Accuracy Estimation** We investigate the effectiveness of various clustering methods for GNN test selection. We extend the concept of model confidence to accuracy estimation, making clustering approaches utilize the model's prediction probability vector for tests (which reflects model confidence) to conduct clustering.
- **Test Selection for Performance Enhancement (using confidence-based approaches)** We investigate the effectiveness of various confidence-based test selection methods, encompassing both approaches for misclassification detection and accuracy estimation, in selecting retraining inputs to enhance the accuracy of GNNs.
- **Test Selection for Performance Enhancement (using approaches based on node importance)** We investigate the effectiveness of node importance-based test selection methods in selecting retraining inputs to improve GNN accuracy. This exploration is motivated by three factors: 1) Nodes with high importance typically encapsulate critical information and exert a more pronounced influence over the entire graph. Therefore, these nodes are more likely to capture essential information crucial for enhancing model performance [270]; 2) Unimportant nodes can contain noise or irrelevant data that can introduce interference during retraining, thereby diminishing model performance; 3) Node importance is a unique data feature in GNNs that can be leveraged for selecting crucial tests. To the best of our knowledge, there has been limited or no study investigating whether node importance can be effectively used for selecting retraining inputs, highlighting the necessity of conducting relevant research.

Building upon these four critical aspects, we perform an empirical study that encompasses 7 graph datasets and 8 GNN models, systematically evaluating the performance of 22 test selection approaches. To offer a more comprehensive evaluation, we incorporate not only node classification datasets [186] but also graph classification datasets [31, 271, 272] in our analysis. Our empirical findings reveal that while certain test selection methods demonstrate efficacy in the context of DNNs [46], they do not translate to the same level of effectiveness when applied to GNNs. We delve into the underlying reasons for this disparity in the experimental section. To provide a concise summary, we present the following key conclusions.

- **Test Selection for Misclassification Detection** In the context of GNNs, confidence-based test selection methods do not exhibit the same level of effectiveness as observed in DNNs.
- **Test Selection for Accuracy Estimation** In most cases, clustering-based test

selection methods that utilize the model's confidence vector perform better than random selection. However, their improvements compared to random selection are slight.

- **Test Selection for Performance Enhancement (using confidence-based approaches)** The effectiveness of both confidence-based and clustering-based test selection methods shows only slight enhancements when compared to random selection in selecting retraining inputs to improve GNN accuracy, despite some methods having been demonstrated as performing well in DNNs [47].
- **Test Selection for Performance Enhancement (using node importance-based approaches)** Node importance-based test selection methods are not suitable for selecting retraining data to improve GNN accuracy, and in many cases, they even perform worse than random selection.

Our empirical study provides valuable insights for engineers seeking to apply test selection metrics in GNN contexts. We emphasize the constraints of current test selection approaches for GNNs, thus providing guidance for future research to develop new approaches tailored for GNNs. Our datasets, results, and tools are accessible to the community on GitHub[1].

In summary, we make the following contributions in this paper:

- We conduct an empirical study to assess the effectiveness of confidence-based test selection methods in identifying potentially misclassified test inputs for GNNs. Our study reveals that confidence-based test selection methods, which perform well in DNNs, do not demonstrate the same level of effectiveness.
- We empirically investigate the effectiveness of clustering approaches that utilize model confidence vectors in estimating GNN accuracy. We demonstrate that clustering-based methods, while consistently performing better than random selection, provide only slight improvements.
- We investigate the effectiveness of misclassification detection approaches and accuracy estimation approaches in selecting retraining inputs to improve GNN accuracy. We find that test selection methods, such as confidence-based and clustering-based test selection methods, demonstrate only slight effectiveness.
- We investigate the effectiveness of test selection methods based on node importance in selecting retraining inputs to improve the GNN accuracy. The results show that node importance-based test selection methods are not suitable, and in many cases, they even perform worse than random selection.

7.2 Background

In this section, we present the fundamental domain concepts central to our research. These encompass Graph Neural Networks, Test Selection in DNN Testing, and Active Learning.

7.2.1 Graph Neural Networks

Graph Neural Networks (GNNs) have demonstrated remarkable effectiveness in addressing machine learning challenges associated with graph-structured data [177, 157, 178]. These challenges span a variety of domains, including social networks [158, 159, 160], recommendation systems [155, 156, 157] and bioinformatics [184, 273, 274]. In Figure 7.1, we present a general pipeline for GNN models, which includes four main

[1]https://github.com/BlueBerry-xueqi/graph_testing

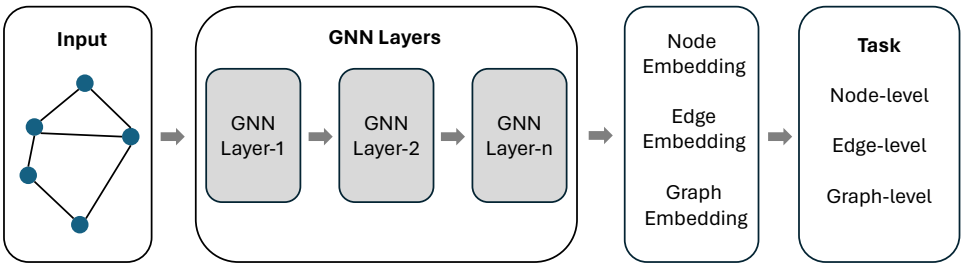


Figure 7.1: The general pipeline for GNN models

parts: 1) The GNN model receives graph-structured inputs, which can contain nodes and edges (representing the connections between nodes). 2) GNN layers then process this graph-structured data. 3) After multiple layers of processing, the GNN model can generate node/edge/graph embedding vectors. These are low-dimensional vector representations of node/edge/graph, facilitating efficient processing and analysis by GNN models. 4) Utilizing these embedding vectors, the GNN model can address tasks at the node-level, edge-level, or graph-level correspondingly.

In the following, we introduce some fundamental concepts related to GNNs and graph datasets.

Graphs A graph can be formally represented as $G = (V, E)$, with V representing the set of nodes and E denoting the set of edges that establish connections between these nodes [275]. Graphs are widely used in various domains [276, 277, 177]. For instance, in citation networks [185], papers can be represented as nodes linked by citations (edges) and grouped into different categories. In chemistry [278], molecules can be viewed as graphs with atoms as nodes and covalent bonds as edges, simplifying the representation of their 3D structures.

Graph Analytics Tasks GNNs can leverage graph structure and node features to perform various analytics tasks. **1) Node-level classification** [29, 279], such as categorizing nodes into distinct classes, utilizes individual node predictions. Prevalent datasets for such tasks include Cora [30], CiteSeer [30], and PubMed [30]. **2) Graph-level classification** aims to determine entire graph attributes, like predicting molecular properties in a chemical graph. Datasets for these tasks include Mutagenicity [31], NCI1 [280], and MSRC21 [272]. **3) Edge-level classification** focuses on classifying edge types between two given nodes. For example, in biological networks, GNNs can utilize the information of a protein and a small molecule to predict their binding affinity, which is considered as edges within a graph. Datasets for edge classification include: DrugBank [281] and BindingDB [282].

Graph Embeddings [188] offer an approach to diminish the dimensions of nodes, edges, and their related attributes while preserving vital structural information and graph characteristics [283]. In graph embedding, each node or edge is mapped to a vector, typically in low dimensions. This low-dimensional representation effectively captures the relationships and similarities between nodes or edges, enabling more efficient computation and analysis within the vector space.

Message Passing The fundamental concept behind Message Passing in GNNs is to enhance the representation of individual nodes by propagating and aggregating information among neighboring nodes, as described in [269]. For example, when calculating the representation of a node N at time step k, the process involves: 1) Gather information from neighbors: Compute the sum of messages from all neighboring nodes of node N to gather information. 2) Utilize the obtained information: Combine the received messages with the representation of node N at time step $(k-1)$

to compute the representation of node N at time step k.

Applying GNNs in Software Engineering GNNs can be applied to various aspects of the field of software engineering. One prevalent application lies in software vulnerability detection [284, 285]. [284] proposed DeepWukong, a novel approach for software vulnerability detection, which utilizes GNNs to encode code fragments into a concise low-dimensional representation. Initially, DeepWukong extracts program slices from code fragments, labeling a slice (or an XFG) as vulnerable if it contains a vulnerable statement. Subsequently, a neural network model is trained using both safe and vulnerable program slices. Both the unstructured and structured code information of a program are incorporated when constructing the neural networks, with both types of information fed into the GNNs to generate a compact code representation in the latent feature space. By leveraging recent advancements in GNNs to learn from vulnerable and safe program slices, DeepWukong enables more precise bug prediction. [285] proposed ContraFlow, which overcomes limitations of previous GNN-based software vulnerability detection methods by focusing on preserving value flow paths rather than the entire graph. By employing contrastive learning, ContraFlow efficiently selects feasible value-flow paths in the embedding space to represent a code fragment accurately. ContraFlow can identify potential error paths based on path-sensitive representations and interpret crucial value flow paths causing vulnerabilities.

7.2.2 Test Selection in DNN Testing

In the context of DNN testing [286, 287, 227, 288], test selection [46, 47] focuses on addressing a practical concern: while collecting unlabelled data is easy and cost-effective, labeling all of it demands substantial effort and specialized domain knowledge. This challenge is typically exacerbated by three key factors: 1) Large-Scale Test Sets: Test sets can be extensive, increasing the workload associated with labeling. 2) Manual Analysis as the Primary Labeling Method: The primary method of labeling involves manual analysis, typically requiring the involvement of multiple individuals to ensure accurate labeling. 3) Dependency on Domain-Specific Knowledge: Labeling frequently necessitates domain-specific expertise, resulting in higher costs associated with employing professionals for the task.

Test selection has emerged as a practical solution for dealing with the labelling cost issue. It involves carefully selecting a subset of unlabeled test data to serve two main objectives: testing DNNs and improving the performance of pre-trained DNNs through retraining. Test selection can be broadly categorized into two main aspects:

- **Misclassification Detection** [46, 3] This aspect focuses on selecting test inputs that are more likely to be misclassified by the DNN model. These tests are more likely to reveal errors in the DNN model and are therefore referred to as “bug-revealing test inputs". Labeling only these test data can lead to reduced overall labeling costs. Furthermore, in active learning contexts, this test data can then be utilized to enhance the model through retraining [47].
- **Accuracy Estimation** [35, 36] This aspect involves selecting a small set of representative test inputs capable of precisely estimating the accuracy of the entire testing dataset. By labeling only these representative tests, it becomes possible to estimate the accuracy of the entire test set, thus reducing labeling costs.

7.2.3 Active Learning

Active learning is a well-established concept within both the software engineering (SE) and machine learning (ML) communities [47]. The fundamental idea behind active learning is to employ machine learning techniques to identify data samples that are relatively challenging to classify [206]. These samples are then presented for human annotation. The annotated data is subsequently used to further train the target ML models to improve the model's performance. The primary objective of active learning is to determine which samples should be prioritized for manual labelling, enabling the model to actively select the informative data to train the model [289]. Existing work [7] has demonstrated that test selection methods can be employed for active learning. [7] empirically investigated the effectiveness of various DNN test selection techniques (e.g., DeepGini and Entropy) in identifying inputs potentially useful for active learning. Their study shows that DeepGini, along with several uncertainty-based methods, can effectively select informative inputs in the context of active learning.

7.3 Approach

In our study, we assessed a total of 22 approaches, comprising 7 test selection methods for misclassification detection, 5 test selection approaches for accuracy estimation, 7 node importance metrics, and one baseline method (i.e., random selection). We selected these approaches for the following reasons: 1) These approaches are adaptable for the corresponding GNN test selection task. For example, DeepGini, as highlighted in its original paper [3], can be used to identify potentially misclassified test inputs; 2) The selected approaches have demonstrated their effectiveness in the context of DNNs [3, 46]; 3) The authors of these approaches have made their implementations publicly available. Below, we will provide a detailed explanation of the basic logic behind each test selection method.

7.3.1 Misclssification Detection Approaches

We employed a total of 10 test selection methods that can be used to detect potentially misclassified GNN tests. One of the classic methods is DeepGini [3]. Moreover, our empirical study also evaluated several active learning-based test selection strategies [172], including Margin Sampling, Least Confidence, and Entropy. Active learning [47] focuses on maximizing model performance gains with minimal sample labeling. Specifically, it aims to select the most valuable samples within an unlabeled dataset and hand them over to the oracle (e.g., human annotator) for labeling, thereby reducing labeling costs while maintaining the model performance. Below, we provide a detailed introduction to the test selection approaches we evaluated.

- **DeepGini** [3] DeepGini quantifies the uncertainty in a model's prediction for a given test by calculating the Gini score of this test. This score is derived from the model's prediction probability vector for the test. A higher Gini value indicates that the model is more uncertain on the specific test. Therefore, the test is considered more likely to be misclassified. The computation of the Gini score is illustrated in Formula 7.1.

$$G(t) = 1 - \Sigma_{i=1}^{N} p_{t,i}^2 \tag{7.1}$$

  where N represents the number of prediction classes, and $p_{t,i}$ represents the probability that the model will classify the test t into class i.

- **Margin Sampling** [172] Margin sampling is an uncertainty-based active learning strategy. Its core idea is to select samples that the model finds most challenging to classify for labeling. Margin Sampling focuses on the difference in the model's predicted probabilities for the two most confident classes. The smaller this probability gap, the more uncertain the model is about the classification of that sample. The uncertainty score of Margin Sampling is calculated by Formula 7.2.

$$Margin(t) = p_k(t) - p_j(t) \tag{7.2}$$

  where $p_k(t)$ refers to the model's predicted probability for the most confident classification. $p_j(t)$ refers to model's predicted probability for the second most confident classification
- **Least Confidence (LC)** [172] Least Confidence is an active learning strategy based on model uncertainty. Specifically, it selects samples for which the model's prediction is the least confident for labeling. In a classification task, if a model has a low maximum predicted probability value for a specific unlabeled sample, it indicates that the model is highly uncertain about the classification of that sample. The Least Confidence strategy selects such samples for labelling. The score of Least Confidence is computed using Formula 7.3.

$$L(t) = 1 - \max_{i=1:n} p_i(t) \tag{7.3}$$

  where $p_i(t)$ represents the probability of test input t being classified into category i. Hence, $\max_{i=1:n} p_i(t)$ represents the model's predicted probability for the most confident classification.
- **Least Confidence-variant (LC-variant)** [172] In contrast to the Least Confidence metric, which ranks classifications based on the most confident predictions, the Least Confidence-variant model assesses uncertainty by focusing on the model's least confident prediction category. This variant considers that when the difference between the model's prediction probability for the least confident classification and 0 is large, it signifies that the model is more uncertain about this test, and this test is more likely to be misclassified. The formula for this variant is provided in Formula 7.4.

  The rationale behind this variant is rooted in the concept of uncertainty, as discussed in previous studies [3]. Specifically, considering a classifier M capable of classifying test inputs into N categories, when the prediction probability vector of M for a test t is $(\frac{1}{N}, \frac{1}{N}, \ldots, \frac{1}{N})$, it signifies that classifier M is the most certain about this test t. Since the highest value that the model's prediction probability can reach for its least confident classification is $\frac{1}{N}$, when the model's prediction probability for the least confident category is higher, it suggests that the model's confidence for the least confident category approaches $\frac{1}{N}$. This suggests that the model exhibits greater uncertainty when predicting this test case. This test is considered more likely to be misclassified.

$$L(t) = \min_{i=1:n} p_i(t) - 0 \tag{7.4}$$

  where $p_i(t)$ represents the probability of test input t being classified into category i. Hence, $\min_{i=1:n} p_i(t)$ represents the model's predicted probability for the least confident classification.

- **Entropy** [7] Entropy is a commonly used method in active learning. It can measure the uncertainty of a model's predictions for a given sample. The entropy method selects samples by calculating the entropy value of the model's predictions for each unlabeled sample. For a given sample, a high entropy value indicates that the model is highly uncertain about the classification of that sample. Therefore, this strategy tends to select samples with high entropy values for labeling, with the aim of improving the model's performance by adding information from these highly uncertain samples.
- **Multiple-Boundary Clustering and Prioritization (MCP)** [290] MCP is an extension of the Margin Sampling. It begins by dividing the data into distinct "boundary areas" based on the top-2 predicted classes. Then, MCP selects data points from each area based on the Margin. The selected data points are considered to be tests for which the model exhibits a higher degree of uncertainty. These tests are considered more likely to be misclassified.
- **Variance** [46] For a given test case, Variance quantifies the uncertainty in the model's predictions by computing the variance of the model's prediction probabilities for that specific test. A smaller variance suggests that the model exhibits greater uncertainty regarding this test, and this test is considered more likely to be misclassified. The formula for Variance is provided in Formula 7.5.

$$\mathrm{Var}(x) = \frac{1}{N}\sum_{i=1}^{N} \mathrm{var}\left(p_i(t)\right) \tag{7.5}$$

  where N represents the number of test inputs in the test set. where $p_i(t)$ represents the probability of test t being classified into category i.
- **ATS** [291] ATS is the first adaptive test selection method designed for DNNs, which utilizes differences in model outputs to measure the diversity of behaviors of DNN test inputs. The objective of ATS is to select more diverse tests from the candidate set, as these tests can reveal more different faults in the DNN-driven software.
- **GraphPrior** [32] GraphPrior is a test prioritization method specifically designed for GNNs. It utilizes mutation testing to prioritize potentially misclassified test inputs. Specifically, given a test set and a GNN model under testing, GraphPrior generates mutated models based on the original GNN model. GraphPrior assumes that a test input is more likely to be misclassified if it can "kill" many mutated models. Based on this assumption, it identifies and prioritizes possibly misclassified tests.
- **Random selection** [133] Through the baseline random selection, tests are selected randomly from the test set.

7.3.2 Accuracy Estimation Approaches

The aforementioned confidence-based methods rely on the model's prediction probability vector to assess whether a test is prone to being incorrectly predicted. These methods are efficient and consume minimal time since they only use the model's final prediction probability vector and mathematical approaches for estimating uncertainty. Based on existing research [35], clustering is a practical approach for test selection to estimate the accuracy of a test set. Clustering groups similar data points together, allowing for the extraction of representative points from each cluster, which can effectively represent the entire test set. Therefore, we empirically explore

the combination of clustering methods with prediction probability vectors for test selection in the context of graph networks to estimate the accuracy of the test set. Below, we introduce all the clustering methods used in our study.

- **K-Means** [292] K-Means is an unsupervised clustering algorithm. The algorithm initially divides the data into K groups and randomly selects K objects as the initial cluster centers. It then computes distances between each point and all the cluster centers, assigning each point to the closest center. Subsequently, the algorithm recalculates the centroid of each cluster. This process continues to iterate until a specific termination condition is met.
- **K-Means Plus** [293] K-Means Plus is an extension of the K-Means algorithm, primarily enhancing the way initial cluster centers are chosen. In the traditional K-Means algorithm, initial cluster centers are randomly chosen, which can lead to different results in different runs and can affect the algorithm's convergence speed and clustering quality. K-Means++ addresses this issue by intelligently selecting the initial cluster centers, aiming to enhance the algorithm's performance.
- **MiniBatch K-means** [294] MiniBatch K-means is an optimized variant of the K-Means algorithm designed for efficiently handling large-scale data, reducing computational time. It utilizes mini-batches, which are small, random, fixed-size data subsets, to manage data in memory. During each iteration, the algorithm gathers a random sample of the data and employs it to update the clusters.
- **Gaussian Mixture Model (GMM)** [295] The Gaussian Mixture Model is a probabilistic model that posits that all data points are generated by a mixture of finite Gaussian distributions with unknown parameters. It can be thought of as an extension of K-means clustering that incorporates information about the data's covariance structure and potential Gaussian distribution centers.
- **Hierarchical Clustering** [296] Hierarchical Clustering is a versatile clustering algorithm that iteratively combines or divides clusters to create nested structures. The hierarchical organization in Hierarchical Clustering is visualized as a tree, with the root representing the cluster containing all samples and the leaves representing clusters with only one sample each.

7.3.3 Node Importance metrics

In RQ4, we employed seven approaches to measure node importance in order to perform test selection. These methods were extracted from existing studies [297, 298, 299, 300].

- **Degree** Degree measures the importance of a node based on the number of edges surrounding the node. Nodes with a higher number of edges are considered more important.
- **Eccentricity** Eccentricity quantifies a node's importance by assessing the longest distance from that node to all other nodes. Nodes with small eccentricity values are deemed more crucial, as they play a pivotal role in connecting various components and influencing information dissemination.
- **Center** The Center approach assesses the importance of a node by calculating its distance from the network center. Nodes closer to the center are considered more important. Center posits that nodes closer to the center have a greater influence and significance in terms of network connectivity and information propagation.
- **Betweenness Centrality (BC)** BC assesses the importance of a node by evaluating its role as an intermediary within the network. The node's betweenness

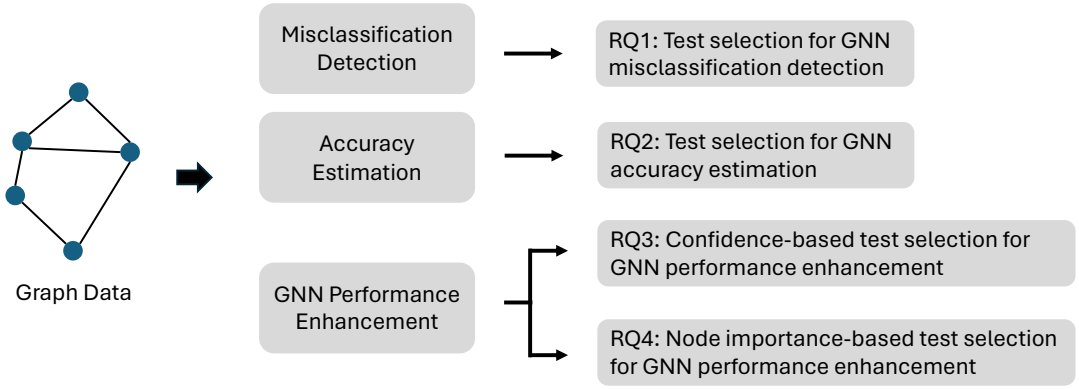


Figure 7.2: Overview of our empirical study

centrality depends on the number of times it acts as a transit point along the shortest paths in the network. A higher betweenness centrality indicates that the node plays a more crucial role in connecting paths between different nodes in the network, and therefore, it is considered more important.

- **Eigenvector Centrality (EC)** Eigenvector Centrality associates a node's importance with the degree to which it is connected to other important nodes. The centrality of a node is determined by the importance of the nodes it is linked to; if a node is connected to others with high Eigenvector Centrality, it will also be considered more important.
- **PageRank** PageRank evaluates the relative significance of nodes in a graph by considering their connectivity and the influence of nodes linked to them. A node's PageRank value depends on both its number of connections and the importance of the nodes that are connected to it. Nodes connected to nodes with higher PageRank values are regarded as more important in this ranking method.
- **Hyperlink-Induced Topic Search (Hits)** Hits determine the importance of nodes through two metrics: Authorities and Hubs. Authorities are assessed based on the quantity and quality of inbound links a node receives, measuring its role as a source of information. Hubs, on the other hand, are evaluated based on the quantity and quality of outbound links, gauging their role as intermediaries in information dissemination. These two metrics interact and are jointly used to assess the relative importance of nodes in the graph network.

7.4 Study design

7.4.1 Overview

Similar to traditional deep neural networks (DNNs), testing Graph Neural Networks (GNNs) also faces challenges due to the absence of automated testing oracles. This leads to the need for manual labeling of test inputs, a process that can be labor-intensive, especially for large and intricate graphs. Furthermore, in specialized domains like drug discovery, as exemplified by protein interface prediction [180], labeling heavily relies on domain-specific knowledge, further escalating costs. In response to the labeling cost issue, existing studies mainly focus on two motivations in the field of DNN testing selection: misclassification detection and accuracy estimation.

- **Misclassification Detection** Misclassification detection aims to select test inputs that are more likely to be misclassified by the DNN model. These selected tests serve two primary purposes: 1) Testers can use them for debugging DNN-based software to enhance the quality of DNNs, and 2) Testers can employ them for

DNN model retraining, effectively reducing the cost associated with retraining.

- **Accuracy Estimation** Accuracy Estimation aims to select a small set of representative test inputs capable of providing an accurate estimate of the entire test set's accuracy.

However, a notable gap exists in adapting DNN test selection methods for GNNs. This challenge emerges due to the distinct nature of GNN test data, where test inputs (nodes) are interconnected, unlike DNNs, where each test sample is treated independently. Consequently, it remains uncertain whether test selection approaches originally tailored for DNNs can be suitably applied to GNNs. To fill the gap, we conduct an empirical study to assess the effectiveness of test selection methods when employed within the context of GNNs, including confidence-based approaches, clustering-based approaches, and node-importance-based approaches.

Figure 7.2 presents an overview of our empirical study. Our study initially focused on three crucial aspects of GNN test selection: GNN accuracy estimation, GNN misclassification detection, and GNN performance enhancement. Specifically, RQ1 focuses on misclassification detection. RQ2 corresponds to accuracy estimation. RQ3 and RQ4 target GNN performance enhancement. In the following, we provide a detailed description of each research question.

- **RQ1: Misclassification Detection.** We evaluate the effectiveness of confidence-based test selection methods for identifying potentially misclassified GNN test inputs, building on their demonstrated efficiency in previous work [3].
- **RQ2: Accuracy Estimation.** We extend the concept of model confidence for accuracy estimation, evaluating the effectiveness of various clustering methods that utilize the model's confidence vector in estimating the accuracy of the GNN test set.
- **RQ3: Performance Enhancement (using confidence-based methods).** We assess the effectiveness of various test selection approaches, encompassing both misclassification detection and accuracy estimation approaches, in selecting retraining inputs for enhancing GNN accuracy.
- **RQ4: Performance Enhancement (using node importance-based methods).** We investigate the effectiveness of node importance-based test selection methods in selecting retraining inputs for improving GNN accuracy. This is motivated by the fact that nodes with high importance tend to capture critical information, while low-importance nodes can introduce noise during retraining. Leveraging node importance in GNNs for test selection is a novel and unexplored area of research.

To provide a more comprehensive assessment, we conducted experiments using a diverse set of 7 graph datasets with 8 GNN models to evaluate the performance of 20 test selection approaches. It is important to emphasize that our dataset includes not only widely adopted node-level datasets but also graph-level datasets in order to ensure a robust evaluation of our methodology. By analyzing the performance of current test selection approaches for GNNs, we aim to investigate the limitations of existing test selection methods in the context of GNNs and provide insights for the future development of novel GNN-oriented test selection methods.

7.4.2 Research Questions

Our experimental evaluation answers the research questions below.

- **RQ1: How effective are different test selection metrics in detecting**

misclassified test inputs for GNNs?
Test selection has emerged as a promising approach for reducing the labelling cost in the testing process. While several test selection techniques have been proposed in the context of DNN testing, their adaptation to GNNs poses distinctive challenges owing to the differences between the test data for DNNs and GNNs. In particular, DNN test inputs are typically independent of one another, whereas GNN test inputs, represented as nodes, exhibit complex interdependencies. Consequently, it remains uncertain whether the DNN test selection methods can perform well on GNNs. In this research question, we assess the effectiveness of multiple test selection approaches in identifying test inputs that are more likely to be misclassified within the context of GNNs.

- **RQ2: How do various accuracy estimation methods perform when applied to GNNs?**
  Test selection approaches for accuracy estimation are designed to select a subset of test inputs that can effectively estimate the accuracy of the entire testing set. By labeling only the selected tests, the labeling costs can be reduced. In this research question, we empirically assess various test selection approaches in estimating the accuracy of GNNs.
- **RQ3: How do different test selection approaches perform in selecting informative inputs for retraining GNN models?**
  In this research question, we investigate the effectiveness of diverse confidence-based test selection methods in selecting retraining inputs for GNN accuracy improvement. These methods encompass misclassification detection approaches (RQ1) and accuracy estimation approaches (RQ2).
- **RQ4: To what extent can node importance guide the selection of retraining inputs for GNNs?**
  In this research question, we explore the effectiveness of node importance-based methods in selecting inputs to enhance GNN accuracy. This exploration is motivated by: 1) Nodes with high importance typically contain crucial information and have a significant impact on the entire graph, making them valuable for improving model performance; 2) Conversely, unimportant nodes can introduce noise or irrelevant data during retraining, potentially degrading model performance; 3) Node importance in GNNs remains unexplored for selecting crucial tests. Our research aims to address this gap.

7.4.3 GNN models and Datasets

In our experiments, we utilized 7 graph datasets and 8 GNN models to assess the performance of 20 test selection approaches. Detailed information about each dataset and model is elaborated upon in the subsequent sections.

7.4.3.1 Graph datasets

To provide a more comprehensive evaluation, our dataset encompasses not only widely adopted node-level datasets but also edge-level and graph-level datasets. Node-level tasks are centered on making predictions for individual nodes within a graph. The node-level datasets we utilized consist of Cora [186], CiteSeer [186], and PubMed [186]. Edge-level datasets focus on predicting edge types between two given nodes. Our adopted edge-level datasets are DrugBank [281] and BindingDB [282]. In contrast, graph-level tasks are oriented towards predicting global properties or characteristics of an entire graph. Our selection of graph-level datasets includes

Mutagenicity [31], NCI1 [280], GraphMNIST [271], and MSRC21 [272].

1) Node Classification Datasets

- **Cora** [186] The Cora dataset comprises 2,708 scientific publications (nodes) and 5,429 links (edges) representing citations between them. Nodes represent machine learning papers, and edges indicate citations between pairs of papers. Each paper is categorized into one of seven classes, including topics like reinforcement learning and neural networks.
- **CiteSeer** [186] The CiteSeer dataset comprises 3,327 scientific publications (nodes) and 4,732 links (edges). Each paper belongs to one of six categories (e.g., artificial intelligence and machine learning).
- **PubMed** [186] The PubMed dataset contains 19,717 diabetes-related scientific publications (nodes) connected by 44,338 links (edges). Publications are classified into three classes (e.g., Cancer and AIDS).

2) Graph Classification Datasets

- **Mutagenicity** [31] The Mutagenicity dataset presents a diverse collection of 4,337 small molecule graphs, each belonging to one of two distinct classes. It serves as a valuable resource for exploring the mutagenic properties of these molecules, offering insights into their potential health and environmental implications.
- **NCI1** [280] NCI1 encompasses 4,110 small molecule graphs, comprising 407 unique molecules classified into two fundamental categories: toxicity and biological relevance. This dataset plays a crucial role in toxicity prediction and drug discovery efforts.
- **GraphMNIST** [271] GraphMNIST stands as a significant resource in the field of computer vision, consisting of a vast database of handwritten digits. It comprises 412 instances across ten distinct classes, corresponding to integer values from 0 to 9.
- **MSRC21** [272] The MSRC21 dataset is a comprehensive compilation of 563 real-world network graphs from the field of computer vision.

3) Edge Classification Datasets

- **DrugBank** [281] The DrugBank dataset is a multi-class classification dataset primarily focused on drug-drug interactions (DDIs). It involves predicting the interaction type between pairs of drugs given their SMILES strings. Compiled manually from FDA/Health Canada drug labels and original literature, the dataset encompasses 86 distinct interaction types, covering a total of 191,808 DDI pairs involving 1,706 unique drugs.
- **BindingDB** [282] BindingDB is a public, web-accessible database dedicated to measuring binding affinities. It primarily focuses on the interactions between proteins considered to be drug targets and drug-like small molecules. In our experiment, we classified edges based on the magnitude of their binding affinities for the edge classification task.

7.4.3.2 GNN models

- **GCN** [163] GCN is a specialized type of convolutional neural network designed to operate directly on graph structures. It addresses the task of classifying nodes within graphs, such as documents in citation networks, where only a limited number of nodes have labels. The fundamental concept behind GCN involves leveraging the relationships between edges in a graph to consolidate node information and produce updated node representations. GCN has found application in various

research studies, as evidenced by its inclusion in prior works [199, 201].

- **GAT** [185] The inception of GAT arose from the necessity to enhance traditional Graph Convolutional Networks (GCN). GCN considers all neighboring nodes as equally important. However, in practical scenarios, different neighboring nodes can hold different degrees of significance. As a result, GAT incorporates a self-attention mechanism that assigns individualized attention scores to each neighbor. Consequently, GAT excels in identifying and prioritizing the most crucial neighbors during the information aggregation process.
- **Graph Isomorphism Network (GIN)** [190] GIN is designed for processing graph data and solving the graph isomorphism problem. Its working principle involves learning the structural information and connectivity patterns among nodes in a graph, enabling effective identification and comparison of isomorphism between different graphs. The core idea of GIN is to iteratively aggregate feature information from nodes within the graph, capturing and representing essential features of the entire graph.
- **Higher-order Graph Neural Networks (GraphNN)** [301] GraphNN is an advanced class of graph-based machine learning models that extend traditional GNNs to capture intricate higher-order relationships within graph-structured data.
- **Message Passing Neural Networks (MPNNs)** [189] MPNN is a general framework for supervised learning on graphs structured data. It is based on the commonness between several state-of-the-art graph-based neural models.
- **Attention-based Graph Neural Network (AGNN)** [302] AGNN is a neural network architecture designed for graph data analysis. Its distinctive feature is the complete removal of traditional fully connected intermediate layers, replaced with attention mechanisms to better preserve the information within the graph structure.
- **Graclus GNNs** [303] Graclus GNNs is an approach that integrates the Graclus graph clustering algorithm with GNNs. Graclus is utilized for partitioning a given graph into clusters or communities based on node similarity or relationships. In this model, the graph data undergoes pre-processing with Graclus.
- **GNNs with convolutional ARMA filters (ARMA)** [271] ARMA refers to an optimized GNN architecture with a new graph convolutional layer inspired by the auto-regressive moving average (ARMA) filter. ARMA brings significant improvements for node classification, graph classification, etc.
- **GSAGE-E** [191] Graph Sample and Aggregate (GraphSAGE) generates embeddings for nodes by accumulating and integrating characteristics from their adjacent nodes. GraphSAGE samples a predetermined quantity of neighbors for each node. GSAGE-E is a variant model of GraphSAGE aimed at edge classification tasks. In this model, the fused information of two nodes (i.e., concatenating the vectors of two nodes) is utilized to predict the category of the edge between them.
- **TAGCN-E** [192] The Topology Adaptive GCN (TAGCN) employs a collection of learnable filters, each of a fixed size, to execute convolutional operations on graph structures. These filters adapt to the unique topology of the graph during the convolution process. TAGCN-E is a variant model of TAGCN that focuses on edge classification. In TAGCN-E, the fused information of two nodes is utilized to predict the category of the edge between them.

7.4.4 Measurements

7.4.4.1 Percentage of Fault Detected (PFD)

Following the prior research [3], we employ PFD to assess the effectiveness of various test selection methods in detecting misclassified test inputs. The computation of PFD is represented in Formula 7.6. From a mathematical standpoint, PFD measures the ratio of correctly detected misclassified test inputs to the total number of misclassified tests within the test set. A higher PFD value indicates that the evaluated test selection approach is more effective at identifying misclassified inputs.

$$PFD = \frac{\#T_{detect}}{\#T_{mis}} \tag{7.6}$$

where $\#T_{detect}$ represents the number of detected misclassified test inputs, while $\#T_{mis}$ denotes the total number of misclassified test inputs in the test set. In our study, we assessed the PFD values of different test selection approaches under varying ratios of prioritized tests

7.4.4.2 Root Mean Square Error

The root mean square error (RMSE) measures the average difference between the estimated accuracy and the actual accuracy of a test set. The calculation formula is shown in Formula 7.7. A lower RMSE value indicates that the selected test inputs can predict the accuracy of the entire test set more accurately, indicating that the utilized test selection method is more effective.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}|\hat{acc}_i - acc|^2} \tag{7.7}$$

where acc refers to the actual accuracy, and $\hat{acc}$ refers to the estimated accuracy.

7.4.5 Implementation and Configuration

This project is implemented using the PyTorch 1.11.0 and PyTorch Geometric 2.1.0 framework. We integrated the available implementations of the test selection approaches [3, 46, 47] into our experimental pipeline. To implement the clustering-based test selection methods, we utilized the package scikit-learn 1.0.2. To implement node importance metrics, we employed the package networkx 2.6.3. Our experiments were conducted on a high-performance computer cluster, with each cluster node equipped with a 2.6 GHz Intel Xeon Gold 6132 CPU and an NVIDIA Tesla V100 16G SXM2 GPU. For data processing tasks, we conducted corresponding experiments on a MacBook Pro laptop running Mac OS Big Sur 11.6, equipped with an Intel Core i9 CPU and 64 GB of RAM.

7.5 Results and analysis

7.5.1 RQ1: Test selection for GNN misclassification detection

Objectives: We investigate the effectiveness of 8 confidence-based test selection methods for GNNs in the context of node classification and graph classification tasks, respectively.

Experimental Design: In the first step, we collected 10 test selection methods from existing studies [46, 3, 7] that can be adapted for GNN misclassification detection.

Table 7.1: Effectiveness of misclassification detection approaches with respect to random selection (baseline) in terms of PFD

Data	Model	Approach								
		DeepGini	ATS	LC	Margin	GraphPrior	Entropy	LC-variant	Variance	MCP
Cora	GCN	4.0588	2.3725	4.1531	**4.7450**	5.9804	3.9332	3.2034	4.0500	4.3557
	GAT	2.0876	2.8715	2.3436	**5.7431**	6.5317	1.9738	1.3559	2.0649	4.5054
	AGNN	3.4733	2.4286	3.4006	**4.8574**	5.6503	3.4509	3.2690	3.4613	3.7715
	ARMA	4.2897	2.7089	4.2859	**5.4179**	6.6693	4.1769	3.5346	4.2250	4.9692
CiteSeer	GCN	1.9209	1.8116	2.1096	**3.6233**	4.9170	1.8282	1.3654	1.8942	3.5128
	GAT	1.0895	2.0998	1.2338	**4.1997**	5.4637	1.0201	0.7090	1.1042	3.5111
	AGNN	2.1020	2.0201	2.0425	**4.0402**	5.2734	2.0891	1.9020	2.0891	3.9655
	ARMA	2.5656	1.8109	2.6523	**3.6219**	4.2475	2.4156	2.0682	2.5614	3.3946
PubMed	GCN	3.1220	2.2013	3.3303	**4.4028**	5.4706	2.8711	1.7775	3.1095	4.2942
	GAT	1.8241	1.5241	1.8955	**5.2141**	5.7141	1.6941	1.2343	1.7783	5.1693
	AGNN	2.5477	2.5891	2.6080	**5.1784**	5.9597	2.4586	1.8991	2.5610	4.8041
	ARMA	4.2553	3.2553	4.2477	5.0853	5.5853	4.1420	2.9595	4.2304	**5.1101**
MSRC21	GraphNN	5.0694	4.0694	5.0944	6.0750	-	4.9306	3.6417	5.1600	4.9083
	GIN	2.5519	2.2519	2.6519	3.9764	-	2.4736	2.2387	2.5962	1.9009
DrugBank	GSAGE-E	4.0417	1.9535	4.0878	3.9071	-	3.8784	3.0482	4.0321	4.0122
	TAGCN-E	3.7352	1.7373	3.7517	3.4748	-	3.5893	2.8072	3.7024	3.6539
BindingDB	GSAGE-E	3.1728	1.5787	3.1799	3.1574	-	3.1616	2.8503	3.1682	3.1701
	TAGCN-E	2.9532	1.4667	2.9856	2.9335	-	2.9582	2.7912	2.9515	2.9574

These approaches have been proven effective in the context of DNNs. Moreover, we also evaluated a test prioritization method specifically designed for GNNs, called GraphPrior [32], and compared its effectiveness with these DNN test prioritization methods. To provide a more comprehensive evaluation, we include not only node classification datasets but also edge classification and graph classification datasets in our analysis. Following the methodology of previous research [3], we utilized the PFD metric to evaluate the effectiveness of various test selection methods in selecting misclassified test inputs. PFD directly measures the ratio of correctly identified misclassified test inputs to the total number of misclassified tests within the test set. Hence, it provides a straightforward reflection of the effectiveness of test selection methods. A higher PFD value indicates that the evaluated test selection approach is more effective at detecting misclassified inputs. Moreover, in order to more clearly demonstrate the difference in effectiveness between the test selection method and the baseline method (random selection), we performed normalization to the experimental results (using Formula 7.8 [304]) and reported the results.

$$x_{\text{normalized}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}} \tag{7.8}$$

where x is the original value. $x_{\min}$ is the minimum value within all the values. $x_{\max}$ is the maximum value within all the values. $x_{\text{normalized}}$ is the resulting normalized value.

Results: The results of RQ1 are presented in Table 7.1, Table 7.2, Table 7.3, and Figure 7.3. Table 7.1 presents the effectiveness of various test selection approaches on graph datasets across three different classification tasks: node classification, edge classification, and graph classification datasets. We shaded the approach with the highest effectiveness for each case in gray. On the node classification dataset, we highlighted in bold the method that performs best among all approaches not specifically designed for GNNs.

From Table 7.1, we see that on the node classification datasets (i.e., Cora, CiteSeer, and PubMed), GraphPrior, specifically designed for GNNs, demonstrates the highest effectiveness across all cases. Furthermore, among all approaches not specifically designed for GNNs, Margin performs the best in the majority of cases (90% among

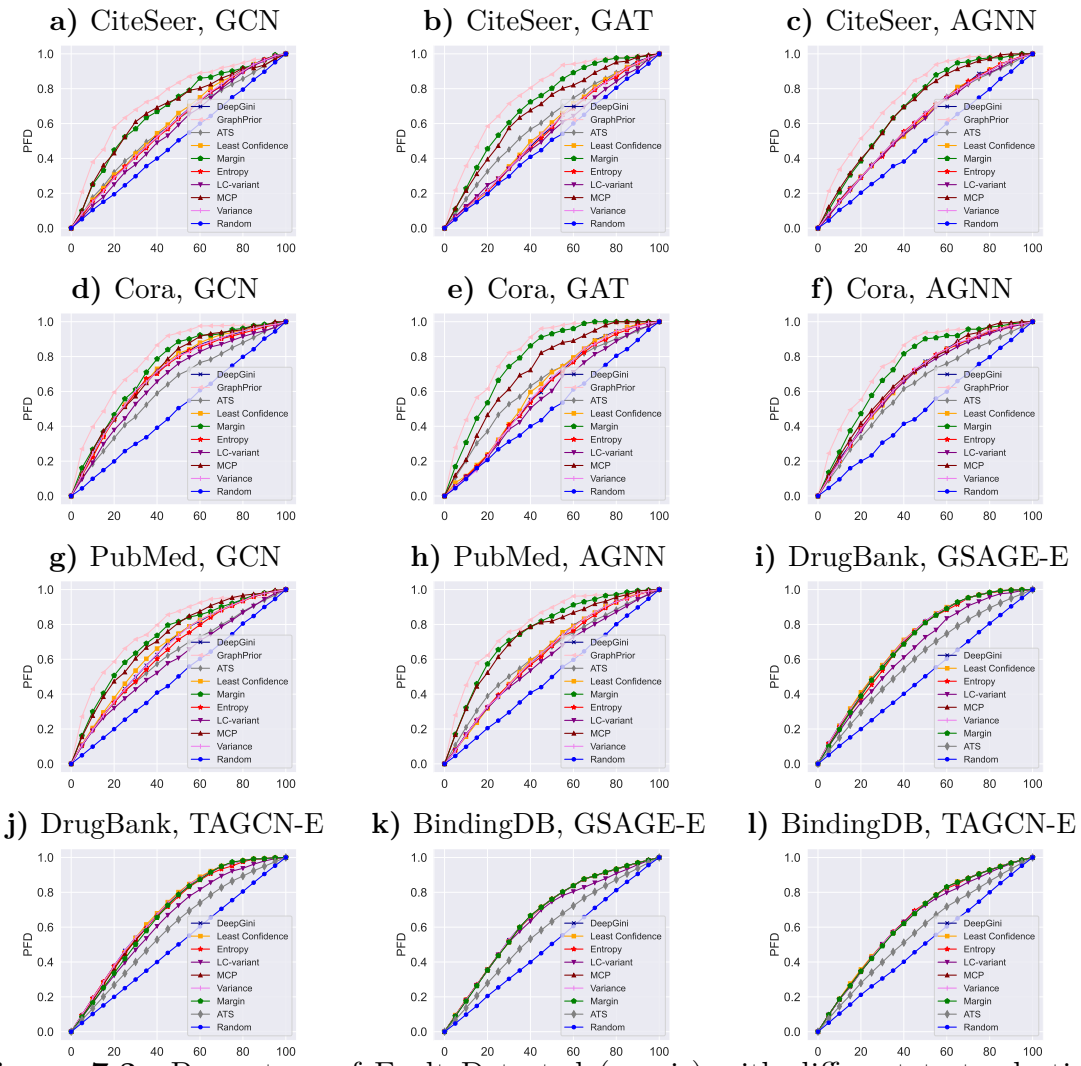


Figure 7.3: Percentage of Fault Detected (y-axis) with different test selection approaches given the ratio of tests executed (x-axis)

Table 7.2: Comparative effectiveness of misclassification detection approaches relative to baseline (normalization analysis)

Data	Model	Approach								
		DeepGini	ATS	LC	Margin	GraphPrior	Entropy	LC-variant	Variance	MCP
Cora	GCN	0.6787	0.3967	0.6945	**0.7934**	1.0000	0.6577	0.5356	0.6772	0.7283
	GAT	0.3196	0.4396	0.3588	**0.8793**	1.0000	0.3022	0.2076	0.3161	0.6898
	AGNN	0.6147	0.4298	0.6018	**0.8597**	1.0000	0.6107	0.5786	0.6126	0.6675
	ARMA	0.6432	0.4062	0.6426	**0.8124**	1.0000	0.6263	0.5300	0.6335	0.7451
CiteSeer	GCN	0.3907	0.3684	0.4290	**0.7369**	1.0000	0.3718	0.2777	0.3852	0.7144
	GAT	0.1994	0.3843	0.2258	**0.7687**	1.0000	0.1867	0.1298	0.2021	0.6426
	AGNN	0.3986	0.3831	0.3873	**0.7661**	1.0000	0.3962	0.3607	0.3962	0.7520
	ARMA	0.6040	0.4263	0.6244	**0.8527**	1.0000	0.5687	0.4869	0.6030	0.7992
PubMed	GCN	0.5707	0.4024	0.6088	**0.8048**	1.0000	0.5248	0.3249	0.5684	0.7850
	GAT	0.3192	0.2667	0.3317	**0.9125**	1.0000	0.2965	0.2160	0.3112	0.9047
	AGNN	0.4275	0.4344	0.4376	**0.8689**	1.0000	0.4125	0.3187	0.4297	0.8061
	ARMA	0.7619	0.5828	0.7605	0.9105	1.0000	0.7416	0.5299	0.7574	**0.9149**
MSRC21	GraphNN	0.8345	0.6699	0.8386	1.0000	-	0.8116	0.5995	0.8494	0.8080
	GIN	0.6418	0.5663	0.6669	1.0000	-	0.6221	0.5630	0.6529	0.4780
DrugBank	GSAGE-E	0.9887	0.4779	1.0000	0.9558	-	0.9488	0.7457	0.9864	0.9815
	TAGCN-E	0.9956	0.4631	1.0000	0.9262	-	0.9567	0.7482	0.9869	0.9739
BindingDB	GSAGE-E	0.9978	0.4965	1.0000	0.9929	-	0.9942	0.8963	0.9963	0.9969
	TAGCN-E	0.9891	0.4913	1.0000	0.9825	-	0.9908	0.9349	0.9886	0.9906

Table 7.3: Effectiveness comparison of misclassification detection approaches on node and graph classification tasks, respectively

Task	**Approach**	**Percentage of test case executed**						
		10%	20%	30%	40%	50%	60%	70%
Node-Level	DeepGini	0.1855	0.3372	0.4792	0.6014	0.7080	0.7971	0.8693
	ATS	0.1833	0.3401	0.4683	0.5805	0.6726	0.7531	0.8212
	LC	0.1850	0.3403	0.4859	0.6154	0.7180	0.8029	0.8743
	Margin	**0.2895**	**0.5031**	**0.6474**	**0.7596**	**0.8366**	**0.8994**	**0.9367**
	GraphPrior	0.3963	0.5812	0.7268	0.8244	0.9015	0.9493	0.9642
	Entropy	0.1827	0.3346	0.4704	0.5928	0.6982	0.7846	0.8640
	LC-variant	0.1750	0.3152	0.4410	0.5543	0.6528	0.7409	0.8203
	MCP	0.2644	0.4669	0.6175	0.7298	0.8119	0.8681	0.9183
	Variance	0.1828	0.3370	0.4784	0.6042	0.7077	0.7966	0.8671
	Random	0.0997	0.2009	0.3014	0.4019	0.4991	0.6033	0.6993
Graph-Level	DeepGini	0.1743	0.3835	0.5691	0.6952	0.7977	0.8614	0.9181
	ATS	0.1723	0.3532	0.4976	0.6401	0.7612	0.8235	0.8991
	LC	0.1878	0.4054	0.5366	0.7131	0.7973	0.8651	0.9257
	Margin	0.2238	0.5498	0.6253	0.7935	0.8684	0.9251	0.9811
	Entropy	0.1781	0.4147	0.5616	0.6934	0.7836	0.8512	0.9013
	LC-variant	0.1653	0.3298	0.4746	0.6033	0.7311	0.8068	0.8906
	MCP	0.2238	0.4193	0.5414	0.6625	0.7558	0.8401	0.8684
	Variance	0.1869	0.4123	0.5818	0.6953	0.7986	0.8693	0.9177
	Random	0.0950	0.2158	0.3005	0.3827	0.4972	0.6088	0.6959
Edge-Level	DeepGini	0.1944	0.3667	0.5318	0.6650	0.7762	0.8602	0.9196
	ATS	0.1414	0.2800	0.4089	0.5288	0.6382	0.7314	0.8115
	LC	0.1936	0.3733	0.5299	0.6689	0.7774	0.8636	0.9194
	Margin	0.1813	0.3560	0.5152	0.6572	0.7730	0.8585	0.9193
	Entropy	0.1934	0.3614	0.5201	0.6613	0.7732	0.8536	0.9156
	LC-variant	0.1795	0.3429	0.4940	0.6197	0.7302	0.8130	0.8784
	MCP	0.1898	0.3654	0.5256	0.6637	0.7755	0.8608	0.9194
	Variance	0.1941	0.3661	0.5312	0.6648	0.7759	0.8575	0.9192
	Random	0.1015	0.2039	0.3026	0.4005	0.5032	0.6043	0.7036

all cases). Similarly, on the graph classification datasets, the best-performing test selection method is also Margin. On the edge classification datasets (i.e., DrugBank and BindingDB), the best-performing method is the least confidence, which performs the best across all cases.

Table 7.2 presents the normalization results for the sum of PFDs for all test selection methods. We utilize random selection as the baseline for normalization. Hence, the normalization results for random selection (baseline) are consistently 0 across all subjects. Detailed normalization calculation methods are provided in the experimental design of RQ1. In this context, if the value for a test selection approach is closer to 1, it indicates that the effectiveness of this test selection method is higher. The experimental results confirm the above conclusions that, on the node classification datasets, GraphPrior, specifically tailored for GNNs, performs as the most effective method in each case. Among the approaches not specifically designed for GNNs, Margin outperforms others in most instances. On graph classification datasets, Margin also performs as the top-performing test selection method. For edge classification datasets, least confidence performs as the most effective approach.

Table 7.3 provides a more detailed breakdown of the effectiveness of various test selection methods across different classification tasks, including the node-level, edge-level, and graph-level classification tasks. The method that performs the best in each case is still highlighted in gray, and in node classification datasets, the best-performing method among all methods not specifically designed for GNNs is also highlighted in bold. In Table 7.3, we see that, in the node classification datasets, the best-performing method is GraphPrior, which is specifically designed for GNNs. Among all methods not specifically designed for GNNs, Margin performs the best. In the edge-level datasets, Least Confidence and DeepGini perform the best. In graph-level datasets, Margin performs the best. This further confirms the conclusions obtained above.

However, we find that in GNNs, uncertainty-based test selection methods (such as Margin Sampling) perform less effectively compared to their performance in the context of traditional DNNs. Based on the findings from previous work (Feng et al., 2020), DeepGini can achieve a PFD of around 90% when selecting 30% of the data, which means that DeepGini can detect about 90% misclassified tests when selecting 30% of tests from the test set. However, as suggested in Figure 7.3, which visually illustrates the effectiveness of different test selection methods, DeepGini can only detect around 50% of misclassified tests when selecting 30% of tests in the context of GNN test selection. Even the best-performing test selection method, Margin Sampling, can only detect approximately 50% to 70% of misclassified tests, significantly lower than its performance on DNNs. Below, we analyze the reasons for the reduced performance of uncertainty-based methods.

There are four potential factors that hinder confidence-based approaches from achieving the same level of effectiveness as in DNNs. In test selection: 1) they do not account for the interdependencies among test inputs (nodes) within the GNN test set, which are crucial for GNN model inference. Confidence-based prioritization approaches typically function on test sets where each test is treated as independent; 2) Irregular Data: Graph data is typically irregular, with varying numbers of connections and neighbor nodes for each node. This irregularity adds complexity to the application of confidence-based approaches to graphs, making it potentially challenging to effectively capture this complexity; 3) Local and Global Dependencies:

Graph data typically exhibit both local and global dependencies. Node attributes and connections can introduce complexity to confidence-based methods since it is challenging to capture these multi-scale dependencies; 4) Size and complexity of graphs. Graphs can exhibit different sizes and complexities. Confidence-based methods can be affected when applied to graph datasets of different sizes and complexities.

Answer to RQ1: *When applied to GNNs (including tasks such as node classification, edge classification, and graph classification), uncertainty-based test selection methods (such as Margin and DeepGini), as well as ATS, do not demonstrate the same level of effectiveness as they exhibited in DNNs.*

7.5.2 RQ2: Test selection for GNN accuracy estimation

Objectives: We evaluate the effectiveness of various clustering methods that utilize the model's confidence vector in estimating the accuracy of the GNN test set.

Experimental Design: In the initial step, we selected five widely recognized clustering algorithms. For each test instance in the test set, we obtain the model's prediction probability vector, which can reflect the model's confidence in its predictions. We call this vector the confidence vector. Following this, we utilize each clustering algorithm to group these instances based on their respective confidence vectors. Subsequently, we select N central points from each cluster. These chosen test instances form a subset of the original test set and can then be employed to predict the overall accuracy of the entire test set.

Table 7.4: Effectiveness of accuracy estimation approaches with respect to random selection (baseline) in terms of RMSE

Data	Model	Approach				
		GMM	Hierarchical	K-Means	K-Means Plus	MiniBatch K-Means
Cora	GCN	0.4332	0.3596	0.3912	0.3713	0.4172
	GAT	-0.0758	0.2509	0.2008	0.0977	-0.0957
	AGNN	0.2029	-0.0571	-0.0479	-0.0273	0.1537
	ARMA	-0.3282	0.1636	0.1449	-0.2729	-0.2430
CiteSeer	GCN	0.2617	-0.2774	-0.2906	0.2151	0.1487
	GAT	0.2181	0.1796	0.2217	0.1153	0.0731
	AGNN	0.2054	0.0359	-0.0089	0.2052	0.2437
	ARMA	0.0860	0.1633	0.1799	0.0878	0.2096
PubMed	GCN	0.2891	0.2037	0.2645	0.3233	0.1979
	GAT	0.2163	0.2303	0.1739	0.2597	0.2321
	AGNN	0.3540	0.3506	0.3078	0.3010	0.3603
	ARMA	0.2070	0.2177	0.1979	0.2435	0.2080
Mutagenicity	GraphNN	0.2364	0.2566	0.2277	0.0892	0.2120
	GIN	0.2773	0.2023	0.2503	0.2190	0.1766
NCI1	GraphNN	0.2582	0.1768	0.2845	0.3240	0.2798
	GIN	0.1849	0.1914	0.1720	0.1341	0.1894
BindingDB	GSAGE-E	-0.0394	-0.0378	-0.0365	-0.0298	-0.0422
	TAGCN-E	0.0076	0.0067	0.0113	0.0191	-0.0013

Results: The results pertaining to RQ2 are presented in Table 7.4, Table 7.5, Table 7.6, and Figure 7.4. Specifically, Table 7.4 exhibits the effectiveness of different test selection approaches related to random selection. In Table 7.4, the values represent the effectiveness of each test selection approach relative to random selection. Specifically, the calculation process is illustrated in Formula 7.9. It is important to note that when using RMSE values to measure effectiveness, a smaller RMSE implies

Table 7.5: Average Effectiveness of accuracy estimation approaches with respect to random selection (baseline) in terms of RMSE

Data	Model	Approach				
		GMM	Hierarchical	K-Means	K-Means Plus	MiniBatch K-Means
Cora	GCN	0.0217	0.0180	0.0196	0.0186	0.0010
	GAT	-0.0002	0.0006	0.0005	0.0002	-0.0002
	AGNN	0.0101	-0.0029	-0.0024	-0.0014	0.0077
	ARMA	-0.0164	0.0082	0.0072	-0.0136	-0.0121
CiteSeer	GCN	0.0131	-0.0139	-0.0145	0.0108	0.0074
	GAT	0.0109	0.0090	0.0111	0.0058	0.0037
	AGNN	0.0103	0.0018	-0.0004	0.0103	0.0122
	ARMA	0.0043	0.0082	0.0090	0.0044	0.0105
PubMed	GCN	0.0145	0.0102	0.0132	0.0162	0.0099
	GAT	0.0108	0.0115	0.0087	0.0130	0.0116
	AGNN	0.0177	0.0175	0.0154	0.0150	0.0180
	ARMA	0.0103	0.0109	0.0099	0.0122	0.0104
Mutagenicity	GraphNN	0.0118	0.0128	0.0114	0.0045	0.0106
	GIN	0.0139	0.0101	0.0125	0.0109	0.0088
NCI1	GraphNN	0.0129	0.0088	0.0142	0.0162	0.0140
	GIN	0.0092	0.0096	0.0086	0.0067	0.0095
BindingDB	GSAGE-E	-0.0020	-0.0019	-0.0018	-0.0015	-0.0021
	TAGCN-E	0.0004	0.0003	0.0006	0.0010	-0.0001

higher effectiveness for a given test selection method. Therefore, in Formula 7.9, if the $diff$ for a test selection method TS is positive, it indicates that the sum of RMSE values for TS is lower than that of random selection, suggesting that the effectiveness of TS is higher than random selection. In the case where TS' $diff$ is positive, if the $diff$ is larger, it indicates that the RMSE values of TS compared to those of random selection are smaller. Since smaller RMSE implies higher effectiveness, it suggests that the effectiveness of TS relative to random selection is higher.

$$diff = \sum_{r=10}^{100} (RMSE^r_{Random} - RMSE^r_{TS}) \tag{7.9}$$

where r represents the number of tests selected. For example, if $r = 80$, it indicates that 80 tests are selected from the test set. $RMSE^i_{TS}$ refers to the effectiveness (measured by RMSE) of the test selection approach TS when selecting r test inputs. $RMSE^i_{Random}$ refers to the effectiveness (measured by RMSE) of random selection when selecting r test inputs.

In Table 7.4, we see that, on node classification datasets (i.e., Cora, CiteSeer, and PubMed), the clustering-based test selection methods perform better than random selection in the majority of cases (85%). Similarly, on graph classification datasets (Mutagenicity and NCI1), the clustering methods consistently perform better than random selection. Table 7.6 further illustrates the effectiveness of different clustering-based test selection approaches, with the best-performing method highlighted in gray for each case. In Table 7.6, the "Number of Selected Test Inputs" indicates the number of tests selected from the test set. We see that, across both node classification and graph classification tasks, clustering-based test selection methods consistently perform the best.

However, the improvement achieved by clustering-based test selection methods compared to random selection is marginal. For example, when selecting ten tests, in terms of RMSE, the best clustering-based method only exceeds random selection by approximately 0.03, and when selecting 80 tests, the best clustering-based method

Table 7.6: Effectiveness comparison among accuracy estimation approaches on node, graph, and edge classification, respectively

Task	Approach	Number of Selected Test Inputs									
		10	20	30	40	50	60	70	80	90	100
Node-Level	GMM	0.0908	0.0719	0.0621	0.0515	0.0473	0.0444	0.0385	0.0406	0.0418	0.0389
	Hierarchical	0.1015	0.0711	0.0598	0.0543	0.0501	0.0384	0.0410	0.0392	0.0363	0.0344
	K-Means	0.1022	0.0739	0.0638	0.0529	0.0456	0.0456	0.0383	0.0370	0.0345	0.0349
	K-Means Plus	0.1041	0.0715	0.0599	0.0510	0.0462	0.0395	0.0399	0.0405	0.0416	0.0386
	MiniBatch K-Means	0.0987	0.0727	0.0624	0.0544	0.0438	0.0467	0.0422	0.0404	0.0338	0.0366
	Random	0.1241	0.0851	0.0724	0.0618	0.0557	0.0475	0.0454	0.0416	0.0439	0.0394
Graph-Level	GMM	0.0850	0.0685	0.0531	0.0464	0.0376	0.0403	0.0359	0.0363	0.0359	0.0281
	Hierarchical	0.1152	0.0544	0.0501	0.0521	0.0555	0.0357	0.0372	0.0333	0.0323	0.0364
	K-Means	0.0942	0.0672	0.0533	0.0556	0.0421	0.0385	0.0335	0.0350	0.0341	0.0358
	K-Means Plus	0.0998	0.0773	0.0537	0.0483	0.0483	0.0410	0.0366	0.0341	0.0423	0.0254
	MiniBatch K-Means	0.0967	0.0765	0.0525	0.0452	0.0417	0.0442	0.0382	0.0369	0.0245	0.0299
	Random	0.1253	0.0897	0.0716	0.0586	0.0550	0.0512	0.0449	0.0407	0.0428	0.0411
Edge-Level	GMM	0.3225	0.1826	0.1810	0.1593	0.1387	0.1294	0.1458	0.0891	0.0792	0.0611
	Hierarchical	0.2662	0.1771	0.1939	0.1854	0.1286	0.1340	0.1397	0.0930	0.0825	0.0722
	K-Means	0.2404	0.1720	0.1885	0.1387	0.1533	0.1550	0.1494	0.1148	0.0744	0.0639
	K-Means Plus	0.1446	0.1631	0.1434	0.1426	0.1507	0.1607	0.1346	0.1365	0.0791	0.0746
	MiniBatch K-Means	0.2126	0.2465	0.1625	0.1558	0.1819	0.1595	0.1657	0.1185	0.0893	0.0852
	Random	0.2326	0.1706	0.1351	0.1164	0.1047	0.0974	0.0903	0.0755	0.0522	0.0517

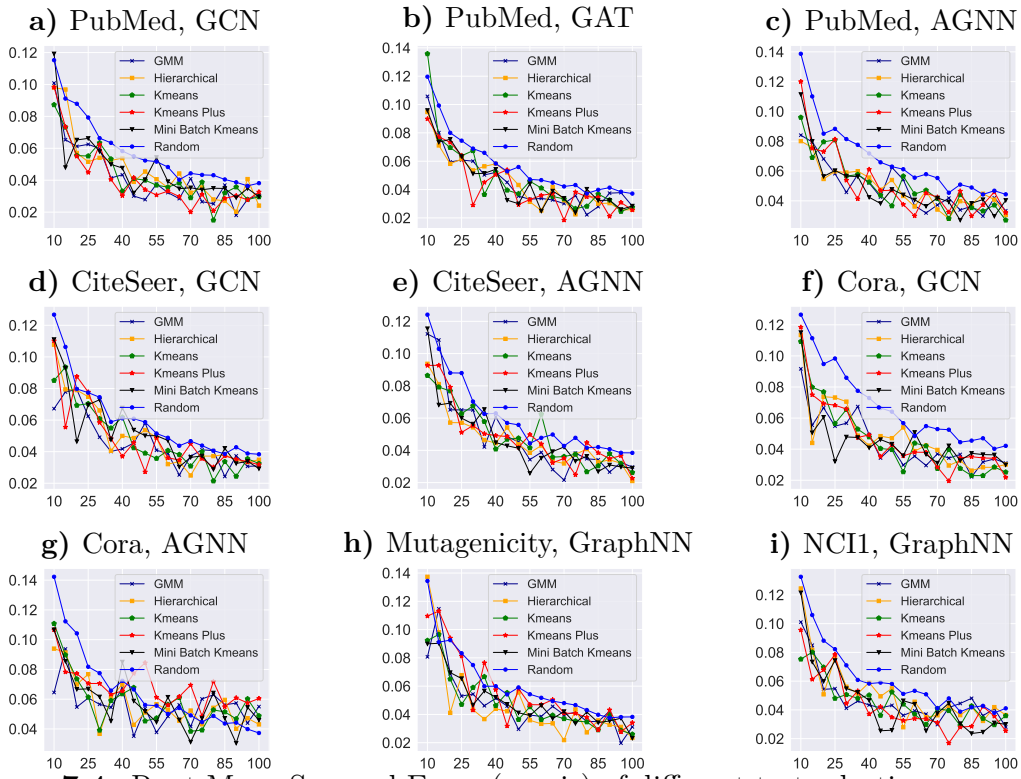


Figure 7.4: Root Mean Squared Errors(y-axis) of different test selection approaches given the number of tests selected (x-axis)

only surpasses random selection by around 0.01. Similarly, Figure 7.4 visually confirms these conclusions, with the blue line representing the baseline (i.e., random selection). We see that while all clustering methods are effective in most cases, the improvements achieved are slight. Moreover, Table 7.5 presents the average differences between all test selection methods and random selection across all cases. The difference values are all around 0.01. Hence, we conclude that clustering-based methods are effective in selecting representative test data for node classification and graph classification datasets but achieve limited improvements.

In the above, we analyzed the effectiveness of the clustering-based test selection approach in node classification and graph classification tasks. Next, we focus on the effectiveness of clustering methods in edge classification datasets (BindingDB). In Table 7.4, we see that, on edge classification datasets, the clustering-based test selection approaches perform better than random selection in 40% of cases. Furthermore, Table 7.6 further highlights the effectiveness of clustering methods across different classification tasks. We see that, in edge-level tasks, random selection exhibits better performance in most cases. From selecting 30 tests and 40 tests up to selecting 100 tests, random selection consistently shows the best performance. This implies that, in the majority of cases, the clustering-based test selection method does not perform as well as random selection on edge classification datasets. In the following, we analyze the potential reasons:

In graph datasets, since a node can be connected to multiple other nodes, the number of edges can far exceed the number of nodes, making the information on edges more complex and diverse, leading to uneven data distribution. Clustering algorithms typically aim to group data points into collections with higher similarities. However, when the data distribution is uneven, clustering algorithms can have difficulty effectively assigning data to the correct clusters. This leads to poor performance when using clustering-based test selection methods in edge classification tasks.

Answer to RQ2: *In node classification and graph classification tasks, all clustering-based test selection methods perform better than random selection in most cases, but their improvements relative to random selection are slight. On edge classification tasks, clustering-based test selection methods do not perform better than random selection in most cases.*

7.5.3 RQ3: Confidence-based test selection for GNN performance enhancement

Objectives: We evaluate the effectiveness of various test selection methods derived from the two aforementioned research questions in selecting informative retraining inputs to enhance GNN model performance. Specifically, these methods correspond to the test selection approaches for misclassification detection (RQ1) and accuracy estimation (RQ2).

Experimental Design: In previous research questions, we assessed multiple test selection methods tailored for misclassification detection and accuracy estimation. In this research question, we apply these methods to select tests for the retraining of the original GNN model, with the objective of improving its prediction accuracy.

The steps and methods we employed for retraining follow the existing study of DNN test selection [46]. In the initial phase, given a GNN model M and a graph dataset, we partition the dataset into a training set, a candidate set, and a test set.

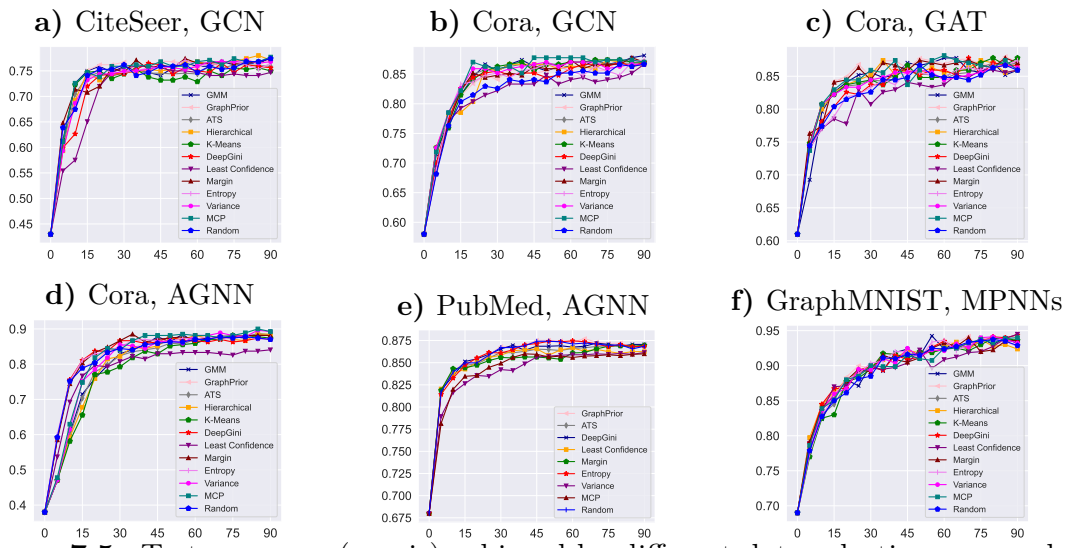


Figure 7.5: Test accuracy (y-axis) achieved by different data selection approaches given the percentage of retrain data selected (x-axis)

Table 7.7: Effectiveness of test selection approaches with respect to random selection (baseline) in selecting retraining inputs to improve GNN accuracy

Data	Model	Approach											
		GMM	Hierarchical	K-Means	Spectrum	GraphPrior	ATS	DeepGini	LC	Margin	Entropy	Variance	MCP
Cora	GCN	0.3741	0.2778	0.3704	**0.4926**	0.2094	0.3759	0.2333	-0.1296	0.2593	0.2259	0.3815	0.4667
	GAT	0.2037	0.2185	0.1926	-0.0333	**0.4004**	0.1629	0.1185	-0.1741	0.3481	0.0444	0.1333	0.2815
	AGNN	-0.3778	-0.4407	-0.6593	-0.5333	0.0449	-0.4426	0.0815	-0.6667	**0.1778**	0.1185	-0.2259	-0.0481
	ARMA	0.0889	0.0111	-0.1296	-0.0111	**0.2561**	-0.0278	0.0815	-0.1407	0.2148	0.0556	0.0741	0.1667
CiteSeer	GCN	0.0482	0.0000	-0.1777	0.0120	0.1191	-0.1009	-0.1777	-0.4518	0.0693	0.0090	-0.0241	**0.1235**
	GAT	-0.0271	0.0090	0.0030	-0.0542	**0.2327**	0.0256	0.0904	-0.1596	0.1837	0.0572	0.0813	0.2319
	AGNN	-0.1536	-0.0723	-0.1084	-0.1265	**-0.0293**	-0.1024	-0.0331	-0.2289	-0.0873	-0.0813	-0.1024	-0.0904
	ARMA	**0.0723**	-0.0331	-0.0663	-0.0873	-0.0654	-0.0632	-0.0151	-0.1265	-0.1114	-0.0843	-0.0361	-0.0512
PubMed	GCN	-	-	-	-	0.1402	0.0426	0.0396	-0.0573	0.0873	0.0553	0.1167	**0.3151**
	GAT	-	-	-	-	0.1057	0.0253	0.0223	0.0061	0.0492	0.007	0.1451	**0.2055**
	AGNN	-	-	-	-	**0.0275**	-0.0621	-0.0147	-0.1096	-0.1243	-0.0096	-0.3125	-0.2760
	ARMA	-	-	-	-	**0.1255**	0.0352	0.0208	-0.0436	0.0680	0.0639	0.0213	0.0685
MNIST	NN	0.0612	0.0750	-0.0012	0.0362	**0.1022**	0.0243	0.0888	-0.0400	0.0562	0.0800	0.0500	0.0175
	GRACLUS	0.2340	0.1180	0.0760	0.2220	0.2099	0.125	0.1300	0.0280	**0.2500**	0.0720	0.1760	-0.0040

The test set remains untouched throughout the process. First, we train an initial GNN model using the training set and record the initial accuracy of M on the test set. Subsequently, we apply various test selection methods to select different subsets of data from the candidate set. We then utilize the selected test data to retrain the original GNN model, recording the model's accuracy after each retraining. By observing the improvement in model accuracy after retraining with data selected using different test selection methods, we can assess and compare the effectiveness of these data selection methods.

Additionally, in the retraining experiments (RQ3), the initial accuracy of the utilized GNN models are: on the CiteSeer dataset: 40% to 60%, on the PubMed dataset: 65% to 70%, on the Cora dataset: 35% to 65%, and on the GraphMNIST dataset: 65% to 70%. The evaluated models' original accuracy range follows the work of [47].

Results: The experimental results for RQ3 are presented in Table 7.7 and Figure 7.5. Table 7.7 presents the effectiveness of all test selection methods in terms of their relative improvement or decline compared to the baseline method (i.e., random selection). We calculate the improvement of each test selection method relative to random selection using Formula 7.10. In Table 7.7, if a test selection method outperforms random, its value is positive and highlighted in gray; conversely, if it performs worse, its value is negative and highlighted in white.

$$imp = \sum_{i=1}^{steps} \left(Acc^{i}_{TS} - Acc^{i}_{Random} \right) \tag{7.10}$$

where *steps* represent the total number of retraining steps. Acc^{i}_{TS} refers to the accuracy of the model retrained using the data selected by the test selection metric TS. Acc^{i}_{Random} refers to the accuracy of the model retrained using the data selected by the random selection approach. *imp* represents the effectiveness improvement of the test selection metric TS over random selection.

In Table 7.7, we see that some test selection methods, such as DeepGini and MCP, along with GraphPrior, perform better than the baseline (random selection) in the majority of cases. Specifically, in approximately 61% of the cases, the evaluated test selection methods perform better than random selection. The top three best-performing methods are GraphPrior, Margin Sampling, and MCP. GraphPrior achieves the best performance in 50% of the cases, MCP in 21.43% of the cases, and Margin Sampling leads in 14.29% of the cases.

However, despite improvements, the extent to which test selection methods improve GNN model accuracy compared to the baseline is slight. Figure 7.5 offers a visual representation of the effectiveness of various test selection methods. In this figure, the blue line denotes the baseline - random selection. We see that the majority of uncertainty-based test selection methods, as well as GraphPrior, only show minor improvements over the baseline (random selection), with some methods even performing worse than random selection. However, a previous study on DNN test selection [47] demonstrated that some uncertainty-based test selection metrics, such as Margin and MCP, can consistently exhibit strong performance. However, these metrics do not achieve consistently strong performance in GNN test selection.

Below, we provide some potential reasons why some test selection methods (e.g., margin and MCP) are effective in DNNs but exhibit only small improvements over the baseline approach when applied to GNNs.

- **Inadequate representativeness** The inputs selected through test selection methods aimed at misclassification detection are typically samples that are more likely to be misclassified. These inputs can be specific in the feature space, representing only a small part of the data distribution. They cannot be sufficient to represent the complex structure and diversity of the entire graph, thus affecting the effectiveness of retraining.
- **Differences in data structure** DNNs typically process data where each sample is independent of others, and retraining the model does not require considering the relationships between samples. In contrast, in graph data processed by GNNs, nodes (i.e., samples) are interconnected through edges. Therefore, the information of a node depends not only on its own features but also on its neighboring nodes and the overall structure of the graph. When test selection methods from DNNs are applied to GNNs, these methods cannot adequately capture and utilize the complex interdependence of graph data for retraining.
- **Differences in Learning Mechanisms** GNNs update node representations by aggregating information from neighboring nodes, which differs from the working mechanism of DNNs. Therefore, the reason for the misclassification of a node can be not only due to the features of the node itself but could also involve information from its neighboring nodes. Simply selecting these misclassified inputs for retraining ignores the crucial information from their neighbors.

Table 7.8: Effectiveness of node importance-based test selection approaches with respect to random selection (baseline) in selecting retraining inputs to improve GNN accuracy

Approach	Percentage of test case executed							
	10%	20%	30%	40%	50%	60%	70%	80%
BC	0.8032	0.8060	0.8087	0.8100	0.8171	0.8225	0.8250	0.8265
Center	0.8002	0.8035	0.8042	0.8066	0.8090	0.8135	0.8173	0.8209
Degree	0.7999	0.8002	0.8041	0.8108	0.8136	0.8201	0.8209	0.8237
EC	0.7994	0.7995	0.8020	0.8061	0.8117	0.8136	0.8144	0.8201
Eccentricity	0.8034	0.8051	0.8040	0.8069	0.8094	0.8131	0.8171	0.8216
Hits	0.7999	0.7995	0.8011	0.8080	0.8118	0.8132	0.8120	0.8175
PageRank	0.8028	0.8074	0.8091	0.8157	0.8190	0.8203	0.8262	0.8297
Random	0.8029	0.8061	0.8164	0.8189	0.8259	0.8328	0.8399	0.8426

Answer to RQ3: *The effectiveness of both confidence-based and clustering-based test selection methods in improving GNN model accuracy through the selection of retraining data shows only slight enhancements when compared to random selection, despite some methods having been demonstrated to be effective in DNNs.*

7.5.4 RQ4: Node importance-based test selection for GNN performance enhancement

Objectives: We assess the effectiveness of node importance-based test selection methods in improving GNN accuracy during retraining. This investigation is driven by several factors: 1) Nodes with high importance typically encapsulate critical information and have a more significant impact on the overall graph [270]. Consequently, these nodes are more likely to capture essential information that is crucial for enhancing model performance; 2) Unimportant nodes may contain noise or irrelevant data that could introduce interference during retraining, potentially leading to a reduction in model performance; 3) Node importance is a distinctive data feature in GNNs that can be leveraged for the selection of critical tests. Currently, there is a gap in research regarding whether node importance can effectively guide the selection of retraining inputs. Therefore, it is imperative to conduct relevant studies in this area.

Experimental Design: In the initial step, we evaluated the initial accuracy of the target GNN model. Subsequently, we ranked all tests in the test set by importance, using each node importance metric. Based on each metric, we selected the top important tests, ranging from 10% to 80%, and then proceeded to retrain the original GNN model. We recorded the model's accuracy after each round of retraining.

Results: The experimental results for RQ4 are presented in Table 7.8. Here, we have shaded in gray the approach with the highest effectiveness for each case. We see that, in the majority of cases, test selection methods based on node importance exhibit limitations when selecting inputs for retraining GNN models to improve accuracy. These methods tend to perform less effectively than random selection. Specifically, random selection outperforms node importance-based methods in 75% of the cases. Conversely, node importance-based test selection methods excel in only the remaining 25% of cases. Some potential factors that can lead to the low performance of node importance-based methods include:

- **Lack of Diversity** Node importance methods can select a group of similar or closely related nodes, potentially resulting in a lack of diversity in the selected data.

In contrast, randomly selecting nodes can introduce greater diversity, thereby enhancing the model's ability to generalize.

- **Overfitting** If the nodes selected by node-importance methods are overly specific or concentrated in a particular area, the model can be prone to overfitting to these selected nodes. Randomly selected nodes, on the other hand, can provide a more varied set of information, contributing to mitigate overfitting
- **Noise Tolerance** Occasionally, incorporating some noisy or less significant nodes can potentially enhance the model's robustness. Randomly selected nodes can introduce such beneficial noise.

Answer to RQ4: *Node importance-based test selection methods are not suitable for selecting retraining data to improve GNN accuracy, and in many cases, they even perform worse than random selection.*

7.6 Threats to Validity

Threats to Internal Validity. The internal threats to validity primarily stem from the implementation of the evaluated test selection approaches. To mitigate this threat, we implemented these approaches using the widely adopted PyTorch library and utilized the implementations of the compared approaches as provided by their respective authors. Another internal threat arises from the selection of clustering algorithms. The effectiveness of test selection can be influenced by the performance of the selected clustering algorithm. To mitigate this threat, we utilized established frameworks in our study. We opted for the widely adopted scikit-learn framework [134] to implement the clustering algorithm. Scikit-learn is renowned for its robust performance and extensive user community.

Threats to External Validity. The primary external threats to the validity of our study are closely linked to two key aspects: the GNN models under evaluation and the test datasets used in our research. These factors can significantly impact the generalizability and applicability of our findings. To mitigate these potential threats, we made a conscious effort to include a large and diverse set of subjects (pairs of datasets and models) in our study. These subjects represent different combinations of GNN models and test datasets, ensuring that our analysis covers a wide spectrum of scenarios. Firstly, we recognized the critical role of dataset diversity and comprehensiveness in evaluating the efficacy of test selection approaches. We utilized seven prevalent graph datasets, encompassing not only node classification datasets but also graph classification datasets. This deliberate selection allows us to account for various problem domains, thereby enhancing the robustness and adaptability of our study to a multitude of GNN applications. Beyond dataset diversity, the choice of GNN models is pivotal in gaining insights into how test selection methods interact with different model architectures. To this end, we utilized a set of eight distinct GNN models, each possessing its unique characteristics and capabilities. These models span a spectrum of complexity and sophistication, ranging from simpler models to more advanced ones.

7.7 Related Work

We present the related works from three perspectives: DNN test selection, DNN Testing, and Empirical study on active learning.

7.7.1 DNN Test Selection

To tackle the challenge of labeling costs, test selection [305] has emerged as a practical solution.

In terms of misclassification detection, Ma *et al.* [46] conducted an evaluation of various test selection methods tailored for misclassification detection, including coverage-based, surprise adequacy-based, and confidence-based approaches. Experimental results demonstrated that confidence-based metrics exhibit a robust ability to identify misclassified inputs, surpassing both the surprise adequacy-based and coverage-based test selection approaches. Hu *et al.* [47] conducted an empirical evaluation of 15 active learning metrics to determine their effectiveness in selecting inputs for retraining DNNs. Their research demonstrated that the choice of data selection metrics can significantly influence the quality of the resulting model when using active learning for training.

Kim *et al.* [49] proposed the Surprise Adequacy Criteria (SADL) for DNN test selection. SADL operates by extracting intermediate outputs from both the test and training data of DNNs, treating them as features, and then evaluating the surprise adequacy based on the dissimilarity between these features. In this process, two measurements are utilized: Likelihood-based Surprise Adequacy (LSA) and Distance-based Surprise Adequacy (DSA). LSA employs kernel density estimation to compute the dissimilarity, while DSA directly utilizes Euclidean distance. Despite the effectiveness of SADL in the context of DNNs, SADL cannot be directly applied to GNNs. This is because implementing SADL requires measuring the distance between the targeted test inputs and training inputs. However, their method for measuring distance is specifically designed for image/text data, which cannot be directly applied to graph-structured data.

Wang *et al.* [2] proposed PRIMA for DNN test prioritization, which identified and prioritizes potentially misclassified test inputs based on intelligent mutation analysis. Despite its effectiveness in the context of DNNs, PRIMA is not suitable for GNNs.This is because PRIMA's mutation operators are not adapted to graph-structured data and GNN models.

In terms of accuracy estimation, Li *et al.* [36] introduced the CES (Cross Entropy-based Sampling) method to tackle this challenge. CES accomplishes test selection by minimizing the cross-entropy between the selected subset and the original test set, ensuring that the distribution of the selected test inputs closely matches that of the original test set. Chen *et al.* [35] proposed the PACE, which employs a range of techniques to perform test selection, including clustering, prototype selection, and adaptive random testing. The process begins by categorizing all test inputs into different groups based on their testing characteristics. Subsequently, PACE utilizes the MMD-critic algorithm [37] to identify prototype test inputs from each group. For test inputs that do not fit into any specific group, PACE employs adaptive random testing to select representative tests.

Our empirical study focuses on evaluating test selection approaches across four areas: 1) Misclassification Detection, 2) Accuracy Estimation, 3) Performance Enhancement guided by confidence-based approaches, and 4) Performance Enhancement guided by node importance-based approaches.

Regarding misclassification detection and performance enhancement, our emphasis has been on evaluating confidence-based approaches due to the following reasons: 1) prior studies [46] have demonstrated that confidence-based methods

outperform coverage-based and surprise-based approaches in terms of effectiveness; 2) Confidence-based test selection methods are widely recognized as the most efficient and straightforward to implement [7], with runtime of less than 1 second in most cases.

7.7.2 Deep Neural Network Testing

In addition to test selection, the field of DNN testing [39, 306, 307] encompasses various noteworthy research directions, with one notable focus being the assessment of DNN adequacy. Pei *et al.* [4] introduced the concept of "neuron coverage" as a metric for gauging the comprehensiveness of a test set in terms of its coverage of a DNN model's logic. They employed this metric to propose a white-box testing framework tailored for DNNs. In a subsequent study, Ma *et al.* [5] introduced DeepGauge, a set of coverage criteria designed to evaluate the adequacy of tests applied to DNNs. DeepGauge also placed significant emphasis on neuron coverage as a valuable indicator of test input effectiveness. Additionally, they introduced novel metrics with varying levels of granularity to distinguish between adversarial attacks and legitimate test data. Kim *et al.* [49] contributed to this area by introducing "surprise adequacy" as a measure for testing DL models. This approach evaluates the effectiveness of a test input by quantifying the surprise it generates concerning the training set. Specifically, the surprise of a test input is determined by measuring the difference in the activation values of neurons when exposed to this new test input.

7.7.3 Empirical study on Active Learning

Active learning has been a subject of extensive research in recent years, with empirical studies spanning various domains. Yu *et al.* [308] conducted empirical research that focused on active learning techniques for literature reviews. In their work, they cataloged and refined three state-of-the-art active learning methods derived from evidence-based medicine and legal electronic discovery. This effort led to the development of a novel active learning approach designed for the analysis of large document corpora, incorporating and fine-tuning the most effective active learning algorithms. Chen *et al.* [309] delved into the effectiveness of active learning in the context of word sense disambiguation. They examined the behavior of active learning by considering two fundamental data selection metrics: entropy and margin. Sassano *et al.* [310] explored the practical application of active learning with Support Vector Machines in a challenging natural language processing task, providing insights into its performance in complex scenarios. Furthermore, Weiss *et al.* [7] conducted a comprehensive investigation into various active learning techniques, revealing that confidence-based methods delivered surprisingly strong results when applied to DNNs.

7.8 Conclusion

In this paper, we conducted a comprehensive empirical study to explore the limitations of test selection approaches in the context of GNNs. We totally evaluated 22 test selection approaches based on 7 graph datasets and 8 GNN models. The results reveal that test selection approaches do not exhibit the same level of effectiveness when applied to GNNs in comparison to DNNs. More specifically, we draw the following conclusions: 1) Confidence-based test selection methods, which perform well in DNNs, do not yield the same level of effectiveness in detecting potentially

misclassified tests for GNNs; 2) In the majority of cases, clustering-based test selection methods that utilize the model's confidence vector perform better than random selection. However, their improvements compared to random selection are slight; 3) In terms of performance enhancement, both confidence-based and clustering-based test selection methods show only slight effectiveness; 4) Node importance-based test selection methods are unsuitable for selecting retraining data to enhance GNN accuracy.

8 Conclusion and Future Work

In this chapter, we conclude this dissertation and outline promising directions for future research.

Contents

8.1 Conclusion

This thesis focused on test prioritization for machine learning models, addressing key challenges in specific scenarios: classical machine learning models, graph neural networks (GNNs), and long-text classification. Below, we provide detailed explanations of the test prioritization methods we proposed for each specific scenario.

In the first part, we proposed MLPrior, a test prioritization method tailored for classical machine learning models. The core idea is that test inputs close to the decision boundary and sensitive to feature mutations are more likely to be misclassified. By leveraging model interpretability and feature attributes, MLPrior effectively identifies potentially misclassified test cases. The evaluation results confirmed the effectiveness of MLPrior across diverse classical ML datasets.

In the second part, we presented GraphPrior, a test prioritization approach specifically developed for graph neural networks (GNNs). GraphPrior considers graph-structured dependencies and incorporates novel mutation rules to prioritize test cases that are more likely to reveal model faults. The evaluation results demonstrated that GraphPrior outperformed all compared methods on both natural and adversarial test cases for GNNs.

In the third part, we introduced LongTest, a test prioritization approach designed for long-text classification. LongTest integrates specialized embeddings and contrastive learning to address the complexity of long-text inputs. Experimental results showed that LongTest outperformed existing methods in prioritizing potentially misclassified test cases for long-text datasets.

Finally, we conducted an empirical study to highlight the limitations of applying existing DNN test selection methods to GNN models. In this study, we evaluated 22 test selection methods on 7 graph datasets and 8 GNN models, revealing that DNN test prioritization approaches are less effective for GNNs. This study highlights the necessity of developing tailored methods specifically for GNN models.

8.2 Future Work

In this section, we outline promising directions for future research.

- **Test Prioritization for Large Language Models (LLMs)** As Large Language Models (LLMs), such as GPT [311], are increasingly adopted across diverse fields (e.g., healthcare, education, and customer service), ensuring their accuracy and reliability has become critical. Despite their impressive capabilities, LLMs can also make errors that may lead to serious consequences, especially in high-stakes contexts such as medical diagnosis support and legal advice generation. To address this issue, one of our future research goals is to develop novel test prioritization methods specifically tailored for LLMs. These methods aim to identify and prioritize test inputs (e.g., questions or prompts) that are more likely to trigger incorrect responses from LLMs. By accelerating the detection of such misclassified cases, the debugging process can be expedited, thereby improving LLM testing efficiency and contributing to the refinement and optimization of their performance.
- **Integration of Test Prioritization with Model Retraining** The main goal of test prioritization is to prioritize inputs that models are likely to misclassify so that these inputs can be used to accelerate the debugging process, thereby improving testing efficiency. Moreover, these prioritized inputs can also be used to retrain models to enhance their performance. Therefore, one of our future research

directions is to investigate whether the proposed test prioritization methods in this dissertation can effectively enhance model retraining. This could involve comparing the effectiveness of these methods against traditional test selection techniques (such as uncertainty-based approaches [3, 172] and surprise-based approaches [49]) to evaluate their effectiveness in improving model accuracy.

- **Test Prioritization for Speech Classification Systems** Beyond the specific domains explored in this dissertation, such as classical machine learning classification, graph neural network (GNN) classification, and long-text classification, other scenarios, such as speech classification, are also worthy of attention. The motivation lies in the fact that Automated Speech Recognition (ASR) systems [312], widely used in applications like virtual assistants and transcription services, require thorough testing to ensure accuracy. However, the cost and time involved in collecting and evaluating speech test cases can be significant. Therefore, one of our future research directions involves developing test prioritization methods tailored specifically for ASR systems to reduce the labelling effort and improve testing efficiency.

Research Activities

In this chapter, we present the research activities conducted throughout my Ph.D. journey. Specifically, we outline 1) the papers to which we contributed and 2) the venues where I have served.

List of Papers

Papers included in this dissertation:

- **Xueqi Dang**, Yinghua Li, Mike Papadakis, Jacques Klein, Tegawendé F. Bissyandé, Yves Le Traon. Test input prioritization for Machine Learning Classifiers. IEEE Transactions on Software Engineering (TSE), 50(3), 413-442. Accepted for publication on 05 January 2024.
- **Xueqi Dang**, Yinghua Li, Mike Papadakis, Jacques Klein, Tegawendé F. Bissyandé, Yves Le Traon. GraphPrior: Mutation-based Test Input Prioritization for Graph Neural Networks. ACM Transactions on Software Engineering and Methodology (TOSEM), 33(1), 1-40. Accepted for publication on 24 November 2023.
- **Xueqi Dang**, Yinghua Li, Wei Ma, Yuejun Guo, Qiang Hu, Mike Papadakis, Maxime Cordy, Yves Le Traon. Towards Exploring the Limitations of Test Selection Techniques on Graph Neural Networks: An Empirical Study. Empirical Software Engineering (EMSE), 29(5), 112. Accepted for publication on 22 July 2024.
- **Xueqi Dang**, Yinghua Li, Wendkuuni C. Ouédraogo, Maxime Cordy, Mike Papadakis, Jacques Klein, Tegawendé F. Bissyandé, Yves Le Traon. LongTest: Test Prioritization for Long Text Files. Under Review in IEEE Transactions on Software Engineering (TSE), 2025.

Papers not included in this dissertation:

- Yinghua Li, **Xueqi Dang**, Lei Ma, Jacques Klein, Yves Le Traon, Tegawendé F. Bissyandé. Test Input Prioritization for 3D Point Clouds. ACM Transactions on Software Engineering and Methodology (TOSEM), 33(5), 1-44. Accepted for publication on 04 June 2024.

Services

Journal Referee:

- Automated Software Engineering

Program Committee:

- The 34th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2025, Tool Demonstrations Track)
- The 47th International Conference on Software Engineering (ICSE 2025, Artifact Evaluation Track)

External Reviewer:

- The 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)

Bibliography

[1] M. J. Raihan, M. A.-M. Khan, S.-H. Kee, and A.-A. Nahid, "Detection of the chronic kidney disease using xgboost classifier and explaining the influence of the attributes on the model using shap," *Scientific Reports*, vol. 13, no. 1, p. 6263, 2023.

[2] Z. Wang, H. You, J. Chen, Y. Zhang, X. Dong, and W. Zhang, "Prioritizing test inputs for deep neural networks via mutation analysis," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 397–409, IEEE, 2021.

[3] Y. Feng, Q. Shi, X. Gao, J. Wan, C. Fang, and Z. Chen, "Deepgini: prioritizing massive tests to enhance the robustness of deep neural networks," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 177–188, 2020.

[4] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, pp. 1–18, 2017.

[5] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, *et al.*, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 120–131, 2018.

[6] M. Wicker, X. Huang, and M. Kwiatkowska, "Feature-guided black-box safety testing of deep neural networks," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 408–426, Springer, 2018.

[7] M. Weiss and P. Tonella, "Simple techniques work surprisingly well for neural network test prioritization and active learning (replicability study)," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 139–150, 2022.

[8] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pp. 785–794, 2016.

[9] Y.-Y. Song and L. Ying, "Decision tree methods: applications for classification and prediction," *Shanghai archives of psychiatry*, vol. 27, no. 2, p. 130, 2015.

[10] P. Gogas and T. Papadimitriou, "Machine learning in economics and finance," *Computational Economics*, vol. 57, pp. 1–4, 2021.

[11] M. Hanafy and R. Ming, "Machine learning approaches for auto insurance big data," *Risks*, vol. 9, no. 2, p. 42, 2021.

[12] M. Fatima, M. Pasha, *et al.*, "Survey of machine learning algorithms for disease diagnostic," *Journal of Intelligent Learning Systems and Applications*, vol. 9, no. 01, p. 1, 2017.

[13] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 1–36, 2020.

[14] R. E. Wright, "Logistic regression.," 1995.

[15] I. Rish *et al.*, "An empirical study of the naive bayes classifier," in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, vol. 3, pp. 41–46, Seattle, WA, USA;, 2001.

[16] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[17] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: analysis, applications, and prospects," *IEEE transactions on neural networks and learning systems*, vol. 33, no. 12, pp. 6999–7019, 2021.

[18] L. R. Medsker, L. Jain, *et al.*, "Recurrent neural networks," *Design and Applications*, vol. 5, no. 64-67, p. 2, 2001.

[19] M. Ghassemi, T. Naumann, P. Schulam, A. L. Beam, I. Y. Chen, and R. Ranganath, "A review of challenges and opportunities in machine learning for health," *AMIA Summits on Translational Science Proceedings*, vol. 2020, p. 191, 2020.

[20] F. Rundo, F. Trenta, A. L. di Stallo, and S. Battiato, "Machine learning for quantitative finance applications: A survey," *Applied Sciences*, vol. 9, no. 24, p. 5574, 2019.

[21] W. Samek, G. Montavon, S. Lapuschkin, C. J. Anders, and K.-R. Müller, "Explaining deep neural networks and beyond: A review of methods and applications," *Proceedings of the IEEE*, vol. 109, no. 3, pp. 247–278, 2021.

[22] Y. Li, X. Dang, H. Tian, T. Sun, Z. Wang, L. Ma, J. Klein, and T. F. Bissyandé, "An empirical study of ai techniques in mobile applications," *Journal of Systems and Software*, vol. 219, p. 112233, 2025.

[23] L. Mason, J. Baxter, P. Bartlett, and M. Frean, "Boosting algorithms as gradient descent," *Advances in neural information processing systems*, vol. 12, 1999.

[24] K. Kowsari, K. Jafari Meimandi, M. Heidarysafa, S. Mendu, L. Barnes, and D. Brown, "Text classification algorithms: A survey," *Information*, vol. 10, no. 4, p. 150, 2019.

[25] E. Haddi, X. Liu, and Y. Shi, "The role of text pre-processing in sentiment analysis," *Procedia computer science*, vol. 17, pp. 26–32, 2013.

[26] S. Sharmin and Z. Zaman, "Spam detection in social media employing machine learning tool for text mining," in *2017 13th international conference on signal-image technology & internet-based systems (SITIS)*, pp. 137–142, IEEE, 2017.

[27] N. Jindal and B. Liu, "Review spam detection," in *Proceedings of the 16th international conference on World Wide Web*, pp. 1189–1190, 2007.

[28] X. Chen, P. Cong, and S. Lv, "A long-text classification method of chinese news based on bert and cnn," *IEEE Access*, vol. 10, pp. 34046–34057, 2022.

[29] S. Xiao, S. Wang, Y. Dai, and W. Guo, "Graph neural networks in node classification: survey and evaluation," *Machine Vision and Applications, Springer,*, vol. 33, pp. 1–19, 2022.

[30] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, "Collective classification in network data," *AI magazine, AAAI,*, vol. 29, no. 3, pp. 93–93, 2008.

[31] K. Riesen and H. Bunke, "Iam graph database repository for graph based pattern recognition and machine learning," in *Structural, Syntactic, and Statistical Pattern Recognition: Joint IAPR International Workshop, SSPR & SPR 2008, Orlando, USA, December 4-6, 2008. Proceedings*, pp. 287–297, Springer, 2008.

[32] X. Dang, Y. Li, M. Papadakis, J. Klein, T. F. Bissyandé, and Y. L. Traon, "Graphprior: Mutation-based test input prioritization for graph neural networks," *ACM Transactions on Software Engineering and Methodology*, 2023.

[33] Y. Li, X. Dang, L. Ma, J. Klein, and T. F. Bissyandé, "Prioritizing test cases for deep learning-based video classifiers," *Empirical Software Engineering*, vol. 29, no. 5, p. 111, 2024.

[34] Y. Li, X. Dang, W. Pian, A. Habib, J. Klein, and T. F. Bissyandé, "Test input prioritization for graph neural networks," *IEEE Transactions on Software Engineering*, vol. 50, no. 6, pp. 1396–1424, 2024.

[35] J. Chen, Z. Wu, Z. Wang, H. You, L. Zhang, and M. Yan, "Practical accuracy estimation for efficient deep neural network testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–35, 2020.

[36] Z. Li, X. Ma, C. Xu, C. Cao, J. Xu, and J. Lü, "Boosting operational dnn testing efficiency through conditioning," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 499–509, 2019.

[37] B. Kim, R. Khanna, and O. O. Koyejo, "Examples are not enough, learn to criticize! criticism for interpretability," *Advances in neural information processing systems*, vol. 29, 2016.

[38] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[39] G. Jahangirova and P. Tonella, "An empirical evaluation of mutation operators for deep learning systems," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 74–84, IEEE, 2020.

[40] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for java," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 297–298, 2009.

[41] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *Proceedings of the 25th international symposium on software testing and analysis*, pp. 449–452, 2016.

[42] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, "Effective test generation using pre-trained large language models and mutation testing," *Information and Software Technology*, vol. 171, p. 107468, 2024.

[43] A. B. Sánchez, P. Delgado-Pérez, I. Medina-Bulo, and S. Segura, "Mutation testing in the wild: findings from github," *Empirical Software Engineering*, vol. 27, no. 6, p. 132, 2022.

[44] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang, "Predictive mutation testing," in *Proceedings of the 25th international symposium on software testing and analysis*, pp. 342–353, 2016.

[45] Y. Tian, C. Sun, B. Poole, D. Krishnan, C. Schmid, and P. Isola, "What makes for good views for contrastive learning?," *Advances in neural information processing systems*, vol. 33, pp. 6827–6839, 2020.

[46] W. Ma, M. Papadakis, A. Tsakmalis, M. Cordy, and Y. L. Traon, "Test selection for deep learning systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 2, pp. 1–22, 2021.

[47] Q. Hu, Y. Guo, M. Cordy, X. Xie, W. Ma, M. Papadakis, and Y. Le Traon, "Towards exploring the limitations of active learning: An empirical study," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 917–929, IEEE, 2021.

[48] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, *et al.*, "Deepmutation: Mutation testing of deep learning systems," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 100–111, IEEE, 2018.

[49] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 1039–1049, IEEE, 2019.

[50] L. Ma, F. Juefei-Xu, M. Xue, B. Li, L. Li, Y. Liu, and J. Zhao, "Deepct: Tomographic combinatorial testing for deep learning systems," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 614–618, IEEE, 2019.

[51] S. Dola, M. B. Dwyer, and M. L. Soffa, "Input distribution coverage: Measuring feature interaction adequacy in neural network testing," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 3, pp. 1–48, 2023.

[52] V. Riccio, N. Humbatova, G. Jahangirova, and P. Tonella, "Deepmetis: Augmenting a deep learning test set to increase its mutation score," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 355–367, IEEE, 2021.

[53] Y. Li, X. Dang, H. Tian, T. Sun, Z. Wang, L. Ma, J. Klein, and T. F. Bissyande, "Ai-driven mobile apps: an explorative study," *arXiv preprint arXiv:2212.01635*, 2022.

[54] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[55] D. W. Otter, J. R. Medina, and J. K. Kalita, "A survey of the usages of deep learning for natural language processing," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 2, pp. 604–624, 2020.

[56] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, *et al.*, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pp. 38–45, 2020.

[57] Z. Batmaz, A. Yurekli, A. Bilge, and C. Kaleli, "A review on deep learning for recommender systems: challenges and remedies," *Artificial Intelligence Review*, vol. 52, pp. 1–37, 2019.

[58] J. Zeng, H. Tang, Y. Li, and X. He, "A deep learning model based on sparse matrix for point-of-interest recommendation.," in *SEKE*, pp. 379–492, 2019.

[59] D. V. Carvalho, E. M. Pereira, and J. S. Cardoso, "Machine learning interpretability: A survey on methods and metrics," *Electronics*, vol. 8, no. 8, p. 832, 2019.

[60] D. Yu, Z. Liu, C. Su, Y. Han, X. Duan, R. Zhang, X. Liu, Y. Yang, and S. Xu, "Copy number variation in plasma as a tool for lung cancer prediction using extreme gradient boosting (xgboost) classifier," *Thoracic cancer*, vol. 11, no. 1, pp. 95–102, 2020.

[61] T. W. Cenggoro, B. Mahesworo, A. Budiarto, J. Baurley, T. Suparyanto, and B. Pardamean, "Features importance in classification models for colorectal cancer cases phenotype in indonesia," *Procedia Computer Science*, vol. 157, pp. 313–320, 2019.

[62] B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran, "Killing strategies for model-based mutation testing," *Softw. Test. Verification Reliab.*, vol. 25, no. 8, pp. 716–748, 2015.

[63] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P. Schobbens, and P. Heymans, "Featured model-based mutation analysis," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016* (L. K. Dillon, W. Visser, and L. A. Williams, eds.), pp. 655–666, ACM, 2016.

[64] M. Papadakis, C. Henard, and Y. L. Traon, "Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing," in *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pp. 1–10, IEEE Computer Society, 2014.

[65] X. Gao, J. Zhai, S. Ma, C. Shen, Y. Chen, and Q. Wang, "Fairneuron: improving deep neural network fairness with adversary games on selective neurons," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 921–933, 2022.

[66] Z. Chen, J. M. Zhang, F. Sarro, and M. Harman, "Maat: a novel ensemble approach to addressing fairness and performance bugs for machine learning software," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1122–1134, 2022.

[67] C. Molinier, P. Temple, and G. Perrouin, "Fairpipes: Data mutation pipelines for machine learning fairness," in *2024 IEEE/ACM International Conference on Automation of Software Test (AST)*, pp. 224–234, 2024.

[68] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yogamani, and P. Pérez, "Deep reinforcement learning for autonomous driving: A survey," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 6, pp. 4909–4926, 2021.

[69] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, 2017.

[70] S. Mallat, "Understanding deep convolutional networks," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2065, p. 20150203, 2016.

[71] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*, pp. 818–833, Springer, 2014.

[72] B. Yeo and D. Grant, "Predicting service industry performance using decision tree analysis," *International Journal of Information Management*, vol. 38, no. 1, pp. 288–300, 2018.

[73] M. Sewak, S. K. Sahay, and H. Rathore, "Comparison of deep learning and the classical machine learning algorithm for the malware detection," in *2018 19th IEEE/ACIS international conference on software engineering, artificial intelligence, networking and parallel/distributed computing (SNPD)*, pp. 293–296, IEEE, 2018.

[74] P. Weber, K. V. Carl, and O. Hinz, "Applications of explainable artificial intelligence in finance—a systematic review of finance, information systems, and computer science literature," *Management Review Quarterly*, pp. 1–41, 2023.

[75] C. Chen, K. Lin, C. Rudin, Y. Shaposhnik, S. Wang, and T. Wang, "A holistic approach to interpretability in financial lending: Models, visualizations, and summary-explanations," *Decision Support Systems*, vol. 152, p. 113647, 2022.

[76] A. Adadi and M. Berrada, "Explainable ai for healthcare: from black box to interpretable models," in *Embedded Systems and Artificial Intelligence: Proceedings of ESAI 2019, Fez, Morocco*, pp. 327–337, Springer, 2020.

[77] M. Verdicchio and A. Perin, "When doctors and ai interact: on human responsibility for artificial risks," *Philosophy & Technology*, vol. 35, no. 1, p. 11, 2022.

[78] H. Smith, "Clinical ai: opacity, accountability, responsibility and liability," *Ai & Society*, vol. 36, no. 2, pp. 535–545, 2021.

[79] J. Amann, A. Blasimme, E. Vayena, D. Frey, V. I. Madai, and P. Consortium, "Explainability for artificial intelligence in healthcare: a multidisciplinary perspective," *BMC medical informatics and decision making*, vol. 20, pp. 1–9, 2020.

[80] T. Grote and P. Berens, "On the ethics of algorithmic decision-making in healthcare," *Journal of medical ethics*, 2019.

[81] H. Yan, S. Lin, *et al.*, "New trend in fintech: Research on artificial intelligence model interpretability in financial fields," *Open Journal of Applied Sciences*, vol. 9, no. 10, p. 761, 2019.

[82] K. Suzuki, "Overview of deep learning in medical imaging," *Radiological physics and technology*, vol. 10, no. 3, pp. 257–273, 2017.

[83] D. Shen, G. Wu, and H.-I. Suk, "Deep learning in medical image analysis," *Annual review of biomedical engineering*, vol. 19, pp. 221–248, 2017.

[84] Z. A. Shirazi, C. P. de Souza, R. Kashef, and F. F. Rodrigues, "Deep learning in the healthcare industry: theory and applications," in *Computational intelligence and soft computing applications in healthcare management science*, pp. 220–245, IGI Global, 2020.

[85] R. Shwartz-Ziv and A. Armon, "Tabular data: Deep learning is not all you need," *Information Fusion*, vol. 81, pp. 84–90, 2022.

[86] Y. Wang and T. Wang, "Application of improved lightgbm model in blood glucose prediction," *Applied Sciences*, vol. 10, no. 9, p. 3227, 2020.

[87] L. H. Gilpin, D. Bau, B. Z. Yuan, A. Bajwa, M. Specter, and L. Kagal, "Explaining explanations: An overview of interpretability of machine learning," in *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*, pp. 80–89, IEEE, 2018.

[88] M. A. Hanif, F. Khalid, R. V. W. Putra, S. Rehman, and M. Shafique, "Robust machine learning systems: Reliability and security for deep neural networks," in *2018 IEEE 24th international symposium on on-line testing and robust system design (IOLTS)*, pp. 257–260, IEEE, 2018.

[89] N. Mehrabi, F. Morstatter, N. Saxena, K. Lerman, and A. Galstyan, "A survey on bias and fairness in machine learning," *ACM Computing Surveys (CSUR)*, vol. 54, no. 6, pp. 1–35, 2021.

[90] A. N. Bhagoji, D. Cullina, C. Sitawarin, and P. Mittal, "Enhancing robustness of machine learning systems via data transformations," in *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*, pp. 1–5, IEEE, 2018.

[91] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, no. 4, pp. 544–558, 2011.

[92] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[93] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant, *Applied logistic regression*. John Wiley & Sons, 2013.

[94] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and regression trees*. Routledge, 2017.

[95] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[96] P. Ammann and J. Offutt, *Introducation to Software Testing*. Cambridge University Press, 2008.

[97] P. Delgado-Pérez, I. Habli, S. Gregory, R. Alexander, J. Clark, and I. Medina-Bulo, "Evaluation of mutation testing in a nuclear industry case study," *IEEE Transactions on Reliability*, vol. 67, no. 4, pp. 1406–1419, 2018.

[98] G. Petrovic, M. Ivankovic, B. Kurtz, P. Ammann, and R. Just, "An industrial application of mutation testing: Lessons, challenges, and research directions," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 47–53, IEEE, 2018.

[99] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 2, pp. 99–118, 1996.

[100] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*, vol. 112, pp. 275–378, Elsevier, 2019.

[101] T. Fredriksson, J. Bosch, and H. H. Olsson, "Machine learning models for automatic labeling: A systematic literature review.," *ICSOFT*, pp. 552–561, 2020.

[102] T. Fredriksson, D. I. Mattos, J. Bosch, and H. H. Olsson, "Data labeling: An empirical investigation into industrial challenges and mitigation strategies," in *International Conference on Product-Focused Software Process Improvement*, pp. 202–216, Springer, 2020.

[103] M. Desmond, E. Duesterwald, K. Brimijoin, M. Brachman, and Q. Pan, "Semi-automated data labeling," in *NeurIPS 2020 Competition and Demonstration Track*, pp. 156–169, PMLR, 2021.

[104] J. Wu, C. Ye, V. S. Sheng, Y. Yao, P. Zhao, and Z. Cui, "Semi-automatic labeling with active learning for multi-label image classification," in *Advances in Multimedia Information Processing–PCM 2015: 16th Pacific-Rim Conference on Multimedia, Gwangju, South Korea, September 16-18, 2015, Proceedings, Part I 16*, pp. 473–482, Springer, 2015.

[105] "Making automated data labeling a reality in modern ai," *Accessed*, 2023.

[106] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli, "Evasion attacks against machine learning at test time," in *Machine Learning and Knowledge Discovery in Databases* (H. Blockeel, K. Kersting, S. Nijssen, and F. Železný, eds.), (Berlin, Heidelberg), pp. 387–402, Springer Berlin Heidelberg, 2013.

[107] G. Guo, H. Wang, D. Bell, Y. Bi, and K. Greer, "Knn model-based approach in classification," in *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003. Proceedings*, pp. 986–996, Springer, 2003.

[108] H. Kim and Z. Gu, "A logistic regression analysis for predicting bankruptcy in the hospitality industry," *The Journal of Hospitality Financial Management*, vol. 14, no. 1, pp. 17–34, 2006.

[109] A. Mayr, H. Binder, O. Gefeller, and M. Schmid, "The evolution of boosting algorithms," *Methods of information in medicine*, vol. 53, no. 06, pp. 419–427, 2014.

[110] P. Chen, S. Liu, H. Zhao, and J. Jia, "Gridmask data augmentation," *arXiv preprint arXiv:2001.04086*, 2020.

[111] Q. H. Nguyen, H.-B. Ly, L. S. Ho, N. Al-Ansari, H. V. Le, V. Q. Tran, I. Prakash, and B. T. Pham, "Influence of data splitting on performance of machine learning models in prediction of shear strength of soil," *Mathematical Problems in Engineering*, vol. 2021, pp. 1–15, 2021.

[112] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[113] S. Boughorbel, F. Jarray, and M. El-Anbari, "Optimal classifier for imbalanced data using matthews correlation coefficient metric," *PloS one*, vol. 12, no. 6, p. e0177678, 2017.

[114] "The adult census income dataset," 2017.

[115] "The bank dataset.," 2014.

[116] T. Tazin, M. N. Alam, N. N. Dola, M. S. Bari, S. Bourouis, M. Monirujjaman Khan, *et al.*, "Stroke disease detection and prediction using robust learning approaches," *Journal of healthcare engineering*, vol. 2021, 2021.

[117] G. ÖZSEZER and G. MERMER, "Diabetes risk prediction with machine learning models," *Artificial Intelligence Theory and Applications*, vol. 2, no. 2, pp. 1–9, 2022.

[118] J. Hua, B. Chu, J. Zou, and J. Jia, "Ecg signal classification in wearable devices based on compressed domain," *Plos one*, vol. 18, no. 4, p. e0284008, 2023.

[119] S. Tizpaz-Niari, A. Kumar, G. Tan, and A. Trivedi, "Fairness-aware configuration of machine learning libraries," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 909–920, 2022.

[120] Y. Li, L. Meng, L. Chen, L. Yu, D. Wu, Y. Zhou, and B. Xu, "Training data debugging for the fairness of machine learning software," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 2215–2227, 2022.

[121] H. Zheng, Z. Chen, T. Du, X. Zhang, Y. Cheng, S. Ji, J. Wang, Y. Yu, and J. Chen, "Neuronfair: Interpretable white-box fairness testing through biased neuron identification," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 1519–1531, 2022.

[122] D. Dua and C. Graff, "Uci machine learning repository. university of california, school of information and computer science, irvine, ca (2019)," 2019.

[123] R. Kohavi *et al.*, "Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid.," in *Kdd*, vol. 96, pp. 202–207, 1996.

[124] A. Ogunleye and Q.-G. Wang, "Xgboost model for chronic kidney disease diagnosis," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 17, no. 6, pp. 2131–2140, 2019.

[125] C. Rao, Y. Liu, and M. Goh, "Credit risk assessment mechanism of personal auto loan based on pso-xgboost model," *Complex & Intelligent Systems*, vol. 9, no. 2, pp. 1391–1414, 2023.

[126] M. Mukid, T. Widiharih, A. Rusgiyono, and A. Prahutama, "Credit scoring analysis using weighted k nearest neighbor," in *Journal of Physics: Conference Series*, vol. 1025, p. 012114, IOP Publishing, 2018.

[127] H. Kamel, D. Abdulah, and J. M. Al-Tuwaijari, "Cancer classification using gaussian naive bayes algorithm," in *2019 International Engineering Conference (IEC)*, pp. 165–170, IEEE, 2019.

[128] D. Slack, S. A. Friedler, C. Scheidegger, and C. D. Roy, "Assessing the local interpretability of machine learning models," *arXiv preprint arXiv:1902.03501*, 2019.

[129] Y. Li, J. Wang, and C. Wang, "Systematic testing of the data-poisoning robustness of knn," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023.

[130] S. A. Friedler, C. Scheidegger, S. Venkatasubramanian, S. Choudhary, E. P. Hamilton, and D. Roth, "A comparative study of fairness-enhancing interventions in machine learning," in *Proceedings of the conference on fairness, accountability, and transparency*, pp. 329–338, 2019.

[131] A. Stevens, P. Deruyck, Z. Van Veldhoven, and J. Vanthienen, "Explainability and fairness in machine learning: Improve fair end-to-end lending for kiva," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 1241–1248, IEEE, 2020.

[132] R. J. Lewis, "An introduction to classification and regression tree (cart) analysis," in *Annual meeting of the society for academic emergency medicine in San Francisco, California*, vol. 14, Citeseer, 2000.

[133] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE transactions on software engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[134] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[135] M. Fan, W. Wei, W. Jin, Z. Yang, and T. Liu, "Explanation-guided fairness testing through genetic algorithm," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 871–882, 2022.

[136] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[137] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse, "Adaptive random test case prioritization," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 233–244, IEEE, 2009.

[138] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei, "Prioritizing junit test cases in absence of coverage information," in *2009 IEEE International Conference on Software Maintenance*, pp. 19–28, IEEE, 2009.

[139] P. Tonella, P. Avesani, and A. Susi, "Using the case-based ranking methodology for test case prioritization," in *2006 22nd IEEE international conference on software maintenance*, pp. 123–133, IEEE, 2006.

[140] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, pp. 201–212, 2009.

[141] Y. Lou, D. Hao, and L. Zhang, "Mutation-based test-case prioritization in software evolution," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 46–57, IEEE, 2015.

[142] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 523–534, IEEE, 2016.

[143] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Information and Software Technology*, vol. 52, no. 1, pp. 14–30, 2010.

[144] H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 1, pp. 1–42, 2013.

[145] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Coverage prediction for accelerating compiler testing," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 261–278, 2018.

[146] N. Humbatova, G. Jahangirova, and P. Tonella, "Deepcrime: mutation testing of deep learning systems based on real faults," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 67–78, 2021.

[147] T. Byun, V. Sharma, A. Vijayakumar, S. Rayadurgam, and D. Cofer, "Input prioritization for testing neural networks," in *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pp. 63–70, IEEE, 2019.

[148] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th international conference on software engineering*, pp. 303–314, 2018.

[149] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making: 9th Asian Computing Science Conference. Dedicated to Jean-Louis Lassez on the Occasion of His 5th Birthday. Chiang Mai, Thailand, December 8-10, 2004. Proceedings 9*, pp. 320–329, Springer, 2005.

[150] D. Shin, S. Yoo, M. Papadakis, and D.-H. Bae, "Empirical evaluation of mutation-based test case prioritization techniques," *Software Testing, Verification and Reliability*, vol. 29, no. 1-2, p. e1695, 2019.

[151] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao, "Deepmutation++: A mutation testing framework for deep learning systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1158–1161, IEEE, 2019.

[152] T. Gaudelet, B. Day, A. R. Jamasb, J. Soman, C. Regep, G. Liu, J. B. Hayter, R. Vickers, C. Roberts, J. Tang, *et al.*, "Utilizing graph machine learning within drug discovery and development," *Briefings in bioinformatics*, vol. 22, no. 6, p. bbab159, 2021.

[153] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *Advances in neural information processing systems*, vol. 33, pp. 22118–22133, 2020.

[154] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[155] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 974–983, 2018.

[156] S. Wu, F. Sun, W. Zhang, X. Xie, and B. Cui, "Graph neural networks in recommender systems: a survey," *ACM Computing Surveys*, vol. 55, no. 5, pp. 1–37, 2022.

[157] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *The world wide web conference*, pp. 417–426, 2019.

[158] C. Li, J. Ma, X. Guo, and Q. Mei, "Deepcas: An end-to-end predictor of information cascades," in *Proceedings of the 26th international conference on World Wide Web*, pp. 577–586, 2017.

[159] L. Wu, P. Sun, R. Hong, Y. Fu, X. Wang, and M. Wang, "Socialgcn: An efficient graph convolutional network based model for social recommendation," *arXiv preprint arXiv:1811.02815*, 2018.

[160] J. Yu, H. Yin, J. Li, M. Gao, Z. Huang, and L. Cui, "Enhance social recommendation with adversarial graph convolutional networks," *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[161] C. Shi, M. Xu, Z. Zhu, W. Zhang, M. Zhang, and J. Tang, "Graphaf: a flow-based autoregressive model for molecular graph generation," *arXiv preprint arXiv:2001.09382*, 2020.

[162] P. Bongini, M. Bianchini, and F. Scarselli, "Molecular generative graph neural networks for drug discovery," *Neurocomputing*, vol. 450, pp. 242–252, 2021.

[163] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[164] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[165] S. Geisler, T. Schmidt, H. Şirin, D. Zügner, A. Bojchevski, and S. Günnemann, "Robustness of graph neural networks at scale," *Advances in Neural Information Processing Systems*, vol. 34, pp. 7637–7649, 2021.

[166] J. Ma, S. Ding, and Q. Mei, "Towards more practical adversarial attacks on graph neural networks," *Advances in neural information processing systems*, vol. 33, pp. 4756–4766, 2020.

[167] Y. Ma, S. Wang, T. Derr, L. Wu, and J. Tang, "Attacking graph convolutional networks via rewiring," *arXiv preprint arXiv:1906.03750*, 2019.

[168] N. Pancino, A. Rossi, G. Ciano, G. Giacomini, S. Bonechi, P. Andreini, F. Scarselli, M. Bianchini, and P. Bongini, "Graph neural networks for the prediction of protein-protein interfaces.," in *ESANN*, pp. 127–132, 2020.

[169] Y. Lou, J. Chen, L. Zhang, and D. Hao, "A survey on regression test-case prioritization," in *Advances in Computers*, vol. 113, pp. 1–46, Elsevier, 2019.

[170] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," *Adv. Comput.*, vol. 112, pp. 275–378, 2019.

[171] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017* (S. Uchitel, A. Orso, and M. P. Robillard, eds.), pp. 597–608, IEEE / ACM, 2017.

[172] D. Wang and Y. Shang, "A new active labeling method for deep learning," in *2014 International joint conference on neural networks (IJCNN)*, pp. 112–119, IEEE, 2014.

[173] D. Zügner and S. Günnemann, "Adversarial attacks on graph neural networks via meta learning," *arXiv preprint arXiv:1902.08412*, 2019.

[174] K. Xu, H. Chen, S. Liu, P.-Y. Chen, T.-W. Weng, M. Hong, and X. Lin, "Topology attack and defense for graph neural networks: An optimization perspective," *arXiv preprint arXiv:1906.04214*, 2019.

[175] A. Bojchevski and S. Günnemann, "Adversarial attacks on node embeddings via graph poisoning," in *International Conference on Machine Learning*, pp. 695–704, PMLR, 2019.

[176] Y. Li, W. Jin, H. Xu, and J. Tang, "Deeprobust: A pytorch library for adversarial attacks and defenses," *arXiv preprint arXiv:2005.06149*, 2020.

[177] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[178] C. Sun, A. Shrivastava, C. Vondrick, R. Sukthankar, K. Murphy, and C. Schmid, "Relational action forecasting," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 273–283, 2019.

[179] R. Yin, K. Li, G. Zhang, and J. Lu, "A deeper graph neural network for recommender systems," *Knowledge-Based Systems*, vol. 185, p. 105020, 2019.

[180] K. Jha, S. Saha, and H. Singh, "Prediction of protein–protein interaction using graph neural networks," *Scientific Reports*, vol. 12, no. 1, pp. 1–12, 2022.

[181] H. Zhou, W. Wang, J. Jin, Z. Zheng, and B. Zhou, "Graph neural network for protein–protein interaction prediction: A comparative study," *Molecules*, vol. 27, no. 18, p. 6135, 2022.

[182] W. Jiang and J. Luo, "Graph neural network for traffic forecasting: A survey," *Expert Systems with Applications*, vol. 207, p. 117921, 2022.

[183] C. Chen, K. Li, S. G. Teo, X. Zou, K. Wang, J. Wang, and Z. Zeng, "Gated residual recurrent graph neural networks for traffic prediction," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, pp. 485–492, 2019.

[184] Q. Zhang, K. Yu, Z. Guo, S. Garg, J. J. Rodrigues, M. M. Hassan, and M. Guizani, "Graph neural network-driven traffic forecasting for the connected internet of vehicles," *IEEE Transactions on Network Science and Engineering*, vol. 9, no. 5, pp. 3015–3027, 2021.

[185] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[186] Z. Yang, W. Cohen, and R. Salakhudinov, "Revisiting semi-supervised learning with graph embeddings," in *International conference on machine learning*, pp. 40–48, PMLR, 2016.

[187] B. Rozemberczki and R. Sarkar, "Characteristic functions on graphs: Birds of a feather, from statistical descriptors to parametric models," in *Proceedings of the 29th ACM international conference on information & knowledge management*, pp. 1325–1334, 2020.

[188] H. Cai, V. W. Zheng, and K. C.-C. Chang, "A comprehensive survey of graph embedding: Problems, techniques, and applications," *IEEE transactions on knowledge and data engineering*, vol. 30, no. 9, pp. 1616–1637, 2018.

[189] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International conference on machine learning*, pp. 1263–1272, PMLR, 2017.

[190] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?," *arXiv preprint arXiv:1810.00826*, 2018.

[191] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[192] J. Du, S. Zhang, G. Wu, J. M. Moura, and S. Kar, "Topology adaptive graph convolutional networks," *arXiv preprint arXiv:1710.10370*, 2017.

[193] H. Dai, H. Li, T. Tian, X. Huang, L. Wang, J. Zhu, and L. Song, "Adversarial attack on graph structured data," in *International conference on machine learning*, pp. 1115–1124, PMLR, 2018.

[194] L. Sun, Y. Dou, C. Yang, J. Wang, P. S. Yu, L. He, and B. Li, "Adversarial attack and defense on graph data: A survey," *arXiv preprint arXiv:1812.10528*, 2018.

[195] D. Zügner, A. Akbarnejad, and S. Günnemann, "Adversarial attacks on neural networks for graph data," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 2847–2856, 2018.

[196] L. Zhang, X. Sun, Y. Li, and Z. Zhang, "A noise-sensitivity-analysis-based test prioritization technique for deep neural networks," *arXiv preprint arXiv:1901.00054*, 2019.

[197] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[198] M. Prince, "Does active learning work? a review of the research," *Journal of engineering education*, vol. 93, no. 3, pp. 223–231, 2004.

[199] X. He, K. Deng, X. Wang, Y. Li, Y. Zhang, and M. Wang, "Lightgcn: Simplifying and powering graph convolution network for recommendation," in *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, pp. 639–648, 2020.

[200] L. Yao, C. Mao, and Y. Luo, "Graph convolutional networks for text classification," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, pp. 7370–7377, 2019.

[201] D. Hong, L. Gao, J. Yao, B. Zhang, A. Plaza, and J. Chanussot, "Graph convolutional networks for hyperspectral image classification," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 59, no. 7, pp. 5966–5978, 2020.

[202] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[203] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[204] D. Zhao, H. Wang, K. Shao, and Y. Zhu, "Deep reinforcement learning with experience replay based on sarsa," in *2016 IEEE symposium series on computational intelligence (SSCI)*, pp. 1–6, IEEE, 2016.

[205] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[206] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, B. B. Gupta, X. Chen, and X. Wang, "A survey of deep active learning," *ACM computing surveys (CSUR)*, vol. 54, no. 9, pp. 1–40, 2021.

[207] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733–752, 2006.

[208] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 700–711, IEEE, 2017.

[209] J. Chen, "Learning to accelerate compiler testing," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pp. 472–475, 2018.

[210] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based test case prioritisation: An industrial case study," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pp. 302–311, IEEE, 2013.

[211] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 235–245, 2014.

[212] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on software engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[213] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 650–670, 2014.

[214] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran, "Prioritizing test cases with string distances," *Automated Software Engineering*, vol. 19, no. 1, pp. 65–95, 2012.

[215] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 ieee symposium on security and privacy (sp)*, pp. 39–57, Ieee, 2017.

[216] W. Shen, J. Wan, and Z. Chen, "Munn: Mutation analysis of neural networks," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 108–115, IEEE, 2018.

[217] S. Minaee, N. Kalchbrenner, E. Cambria, N. Nikzad, M. Chenaghlu, and J. Gao, "Deep learning–based text classification: a comprehensive review," *ACM computing surveys (CSUR)*, vol. 54, no. 3, pp. 1–40, 2021.

[218] Q. Li, H. Peng, J. Li, C. Xia, R. Yang, L. Sun, P. S. Yu, and L. He, "A survey on text classification: From traditional to deep learning," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 13, no. 2, pp. 1–41, 2022.

[219] S. Gao, M. Alawad, M. T. Young, J. Gounley, N. Schaefferkoetter, H. J. Yoon, X.-C. Wu, E. B. Durbin, J. Doherty, A. Stroup, *et al.*, "Limitations of transformers on clinical text classification," *IEEE journal of biomedical and health informatics*, vol. 25, no. 9, pp. 3596–3607, 2021.

[220] F. Wei, H. Qin, S. Ye, and H. Zhao, "Empirical study of deep learning for text classification in legal document review," in *2018 IEEE International Conference on Big Data (Big Data)*, pp. 3317–3320, IEEE, 2018.

[221] Y. Zhang, B. Jin, X. Chen, Y. Shen, Y. Zhang, Y. Meng, and J. Han, "Weakly supervised multi-label classification of full-text scientific papers," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 3458–3469, 2023.

[222] C. Che, H. Hu, X. Zhao, S. Li, and Q. Lin, "Advancing cancer document classification with r andom forest," *Academic Journal of Science and Technology*, vol. 8, no. 1, pp. 278–280, 2023.

[223] S. Hu, F. Teng, L. Huang, J. Yan, and H. Zhang, "An explainable cnn approach for medical codes prediction from clinical text," *BMC Medical Informatics and Decision Making*, vol. 21, pp. 1–12, 2021.

[224] N. Salem and S. Hussein, "Data dimensional reduction and principal components analysis," *Procedia Computer Science*, vol. 163, pp. 292–299, 2019.

[225] X. Dong, L. Qian, Y. Guan, L. Huang, Q. Yu, and J. Yang, "A multiclass classification method based on deep learning for named entity recognition in electronic medical records," in *2016 New York scientific data summit (NYSDS)*, pp. 1–10, IEEE, 2016.

[226] H. Hu, X. Wang, Y. Zhang, Q. Chen, and Q. Guan, "A comprehensive survey on contrastive learning," *Neurocomputing*, p. 128645, 2024.

[227] X. Dang, Y. Li, M. Papadakis, J. Klein, T. F. Bissyandé, and Y. Le Traon, "Test input prioritization for machine learning classifiers," *IEEE Transactions on Software Engineering*, 2024.

[228] Y. Li, X. Dang, L. Ma, J. Klein, Y. Le Traon, and T. F. Bissyandé, "Test input prioritization for 3d point clouds," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–44, 2024.

[229] X. Dang, Y. Li, W. Ma, Y. Guo, Q. Hu, M. Papadakis, M. Cordy, and Y. L. Traon, "Towards exploring the limitations of test selection techniques on graph neural networks: An empirical study," *Empirical Software Engineering*, vol. 29, no. 5, p. 112, 2024.

[230] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, "Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers," *Advances in Neural Information Processing Systems*, vol. 33, pp. 5776–5788, 2020.

[231] L. Van Der Maaten, E. O. Postma, H. J. Van Den Herik, *et al.*, "Dimensionality reduction: A comparative review," *Journal of machine learning research*, vol. 10, no. 66-71, p. 13, 2009.

[232] A. Cutler, D. R. Cutler, and J. R. Stevens, "Random forests," *Ensemble machine learning: Methods and applications*, pp. 157–175, 2012.

[233] T. K. Kim, "T test as a parametric statistic," *Korean journal of anesthesiology*, vol. 68, no. 6, pp. 540–546, 2015.

[234] K. Kelley and K. J. Preacher, "On effect size.," *Psychological methods*, vol. 17, no. 2, p. 137, 2012.

[235] I. Chalkidis, M. Fergadiotis, P. Malakasiotis, and I. Androutsopoulos, "Large-scale multi-label text classification on eu legislation," *arXiv preprint arXiv:1906.02192*, 2019.

[236] B. Jikadara, "Fake News Detection," 2023. Accessed: 2024-09-07.

[237] H. Y. Koh, J. Ju, M. Liu, and S. Pan, "An empirical survey on long document summarization: Datasets, models, and metrics," *ACM computing surveys*, vol. 55, no. 8, pp. 1–35, 2022.

[238] V. Sanh, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[239] B. De Ville, "Decision trees," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 5, no. 6, pp. 448–455, 2013.

[240] J. M. Hilbe, "Logistic regression.," *International encyclopedia of statistical science*, vol. 1, pp. 15–32, 2011.

[241] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[242] S. Ö. Arik and T. Pfister, "Tabnet: Attentive interpretable tabular learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 35, pp. 6679–6687, 2021.

[243] K. Song, X. Tan, T. Qin, J. Lu, and T.-Y. Liu, "Mpnet: Masked and permuted pre-training for language understanding," *Advances in neural information processing systems*, vol. 33, pp. 16857–16867, 2020.

[244] M. Abadi, "Tensorflow: learning functions at scale," in *Proceedings of the 21st ACM SIGPLAN international conference on functional programming*, pp. 1–1, 2016.

[245] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[246] N. Reimers and I. Gurevych, "Making monolingual sentence embeddings multilingual using knowledge distillation," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, 11 2020.

[247] J. Devlin, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[248] Z. Yang, "Xlnet: Generalized autoregressive pretraining for language understanding," *arXiv preprint arXiv:1906.08237*, 2019.

[249] A. Vaswani, "Attention is all you need," *Advances in Neural Information Processing Systems*, 2017.

[250] R. Meyes, M. Lu, C. W. de Puiseau, and T. Meisen, "Ablation studies to uncover structure of learned representations in artificial neural networks," in *Proceedings on the International Conference on Artificial Intelligence (ICAI)*, pp. 185–191, The Steering Committee of The World Congress in Computer Science, Computer . . . , 2019.

[251] B. F. Ljungberg, "Dimensionality reduction for bag-of-words models: Pca vs lsa," *Semanticscholar. org*, 2019.

[252] D. Sachin *et al.*, "Dimensionality reduction and classification through pca and lda," *International journal of computer Applications*, vol. 122, no. 17, 2015.

[253] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Software Engineering*, vol. 19, pp. 182–212, 2014.

[254] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pp. 179–188, IEEE, 1999.

[255] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system," *Software Testing, Verification and Reliability*, vol. 25, no. 4, pp. 371–396, 2015.

[256] M. Pezze, *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[257] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "Autoblacktest: Automatic black-box testing of interactive applications," in *2012 IEEE fifth international conference on software testing, verification and validation*, pp. 81–90, IEEE, 2012.

[258] M. Brunetto, G. Denaro, L. Mariani, and M. Pezzè, "On introducing automatic test case generation in practice: A success story and lessons learned," *Journal of Systems and Software*, vol. 176, p. 110933, 2021.

[259] J. Chen, Y. Lou, L. Zhang, J. Zhou, X. Wang, D. Hao, and L. Zhang, "Optimizing test prioritization via test distribution analysis," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 656–667, 2018.

[260] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: a systematic literature review," *Empirical Software Engineering*, vol. 27, no. 2, p. 29, 2022.

[261] Z. Chen, J. Chen, W. Wang, J. Zhou, M. Wang, X. Chen, S. Zhou, and J. Wang, "Exploring better black-box test case prioritization via log analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 3, pp. 1–32, 2023.

[262] V. Riccio and P. Tonella, "Model-based exploration of the frontier of behaviours for deep learning system testing," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 876–888, 2020.

[263] Z. Tahereh, V. Riccio, T. Paolo, *et al.*, "Deepatash: Focused test generation for deep learning systems," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023.

[264] V. Riccio and P. Tonella, "When and why test generators for deep learning produce invalid inputs: an empirical study," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1161–1173, IEEE, 2023.

[265] T. Zohdinasab, V. Riccio, A. Gambi, and P. Tonella, "Efficient and effective feature space exploration for testing deep learning systems," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–38, 2023.

[266] M. Biagiola and P. Tonella, "Boundary state generation for testing and improvement of autonomous driving systems," *IEEE Transactions on Software Engineering*, 2024.

[267] L. Zhao, T. Zhao, Z. Lin, X. Ning, G. Dai, H. Yang, and Y. Wang, "Flasheval: Towards fast and accurate evaluation of text-to-image diffusion generative models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 16122–16131, 2024.

[268] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional networks on graphs for learning molecular fingerprints," *Advances in neural information processing systems, ACM New York, NY,*, vol. 28, 2015.

[269] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[270] N. Park, A. Kan, X. L. Dong, T. Zhao, and C. Faloutsos, "Estimating node importance in knowledge graphs using graph neural networks," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 596–606, ACM New York, NY, 2019.

[271] F. M. Bianchi, D. Grattarola, L. Livi, and C. Alippi, "Graph neural networks with convolutional arma filters," *IEEE transactions on pattern analysis and machine intelligence, IEEE,*, vol. 44, no. 7, pp. 3496–3507, 2021.

[272] M. Neumann, R. Garnett, C. Bauckhage, and K. Kersting, "Propagation kernels: efficient graph kernels from propagated information," *Machine learning, Springer,*, vol. 102, pp. 209–245, 2016.

[273] Y. Long, M. Wu, Y. Liu, Y. Fang, C. K. Kwoh, J. Chen, J. Luo, and X. Li, "Pre-training graph neural networks for link prediction in biomedical networks," *Bioinformatics, Oxford University Press,*, vol. 38, no. 8, pp. 2254–2262, 2022.

[274] M. Réau, N. Renaud, L. C. Xue, and A. M. Bonvin, "Deeprank-gnn: a graph neural network framework to learn patterns in protein–protein interfaces," *Bioinformatics, Oxford University Press,*, vol. 39, no. 1, p. btac759, 2023.

[275] V. P. Dwivedi, C. K. Joshi, A. T. Luu, T. Laurent, Y. Bengio, and X. Bresson, "Benchmarking graph neural networks," *arXiv preprint arXiv:2003.00982*, 2020.

[276] M. Liu, H. Gao, and S. Ji, "Towards deeper graph neural networks," in *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 338–348, ACM New York, NY, 2020.

[277] W. Jin, Y. Ma, X. Liu, X. Tang, S. Wang, and J. Tang, "Graph structure learning for robust graph neural networks," in *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 66–74, ACM New York, NY, 2020.

[278] O. Wieder, S. Kohlbacher, M. Kuenemann, A. Garon, P. Ducrot, T. Seidel, and T. Langer, "A compact review of molecular property prediction with graph neural networks," *Drug Discovery Today: Technologies, Elsevier,*, vol. 37, pp. 1–12, 2020.

[279] T. Zhao, X. Zhang, and S. Wang, "Graphsmote: Imbalanced node classification on graphs with graph neural networks," in *Proceedings of the 14th ACM international conference on web search and data mining*, pp. 833–841, ACM New York, NY, 2021.

[280] N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels.," *Journal of Machine Learning Research, JMLR,*, vol. 12, no. 9, 2011.

[281] D. S. Wishart, Y. D. Feunang, A. C. Guo, E. J. Lo, A. Marcu, J. R. Grant, T. Sajed, D. Johnson, C. Li, Z. Sayeeda, *et al.*, "Drugbank 5.0: a major update to the drugbank database for 2018," *Nucleic acids research, Oxford University Press*, vol. 46, no. D1, pp. D1074–D1082, 2018.

[282] T. Liu, Y. Lin, X. Wen, R. N. Jorissen, and M. K. Gilson, "Bindingdb: a web-accessible database of experimentally determined protein–ligand binding affinities," *Nucleic acids research, Oxford University Press*, vol. 35, no. suppl_1, pp. D198–D201, 2007.

[283] X. Fu, J. Zhang, Z. Meng, and I. King, "Magnn: Metapath aggregated graph neural network for heterogeneous graph embedding," in *Proceedings of The Web Conference 2020*, pp. 2331–2341, ACM New York, NY, 2020.

[284] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Transactions on Software Engineering and Methodology (TOSEM), ACM New York, NY, USA*, vol. 30, no. 3, pp. 1–33, 2021.

[285] X. Cheng, G. Zhang, H. Wang, and Y. Sui, "Path-sensitive code embedding via contrastive learning for software vulnerability detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 519–531, ACM New York, NY, USA, 2022.

[286] F. U. Haq, D. Shin, S. Nejati, and L. Briand, "Can offline testing of deep neural networks replace their online testing? a case study of automated driving systems," *Empirical Software Engineering, Springer,*, vol. 26, no. 5, p. 90, 2021.

[287] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.

[288] Y. Li, X. Dang, L. Ma, J. Klein, Y. L. Traon, and T. F. Bissyandé, "Test input prioritization for 3d point clouds," *ACM Transactions on Software Engineering and Methodology, ACM New York, NY*, 2023.

[289] H. Ranganathan, H. Venkateswara, S. Chakraborty, and S. Panchanathan, "Deep active learning for image classification," in *2017 IEEE International Conference on Image Processing (ICIP)*, pp. 3934–3938, IEEE, 2017.

[290] W. Shen, Y. Li, L. Chen, Y. Han, Y. Zhou, and B. Xu, "Multiple-boundary clustering and prioritization to promote neural network retraining," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 410–422, IEEE, 2020.

[291] X. Gao, Y. Feng, Y. Yin, Z. Liu, Z. Chen, and B. Xu, "Adaptive test selection for deep neural networks," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 73–85, IEEE, 2022.

[292] M. Ahmed, R. Seraj, and S. M. S. Islam, "The k-means algorithm: A comprehensive survey and performance evaluation," *Electronics, MDPI,*, vol. 9, no. 8, p. 1295, 2020.

[293] D. Arthur and S. Vassilvitskii, "K-means++ the advantages of careful seeding," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 1027–1035, ACM New York, NY, USA, 2007.

[294] D. Sculley, "Web-scale k-means clustering," in *Proceedings of the 19th international conference on World wide web*, pp. 1177–1178, ACM New York, NY, 2010.

[295] E. Patel and D. S. Kushwaha, "Clustering cloud workloads: K-means vs gaussian mixture model," *Procedia computer science, Elsevier,*, vol. 171, pp. 158–167, 2020.

[296] M. Kaushik and B. Mathur, "Comparative study of k-means and hierarchical clustering techniques," *International Journal of Software & Hardware Research in Engineering, iJournals,*, vol. 2, no. 6, pp. 93–98, 2014.

[297] P. Hu, W. Fan, and S. Mei, "Identifying node importance in complex networks," *Physica A: Statistical Mechanics and its Applications, Elsevier,*, vol. 429, pp. 169–176, 2015.

[298] Q. Qiong and W. Dongxia, "Evaluation method for node importance in complex networks based on eccentricity of node," in *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, pp. 2499–2502, IEEE, 2016.

[299] Y. Yang, L. Yu, X. Wang, Z. Zhou, Y. Chen, and T. Kou, "A novel method to evaluate node importance in complex networks," *Physica A: Statistical Mechanics and its Applications, Elsevier,*, vol. 526, p. 121118, 2019.

[300] H. Ando, M. Bell, F. Kurauchi, K.-I. Wong, and K.-F. Cheung, "Connectivity evaluation of large road network by capacity-weighted eigenvector centrality analysis," *Transportmetrica A: Transport Science, Taylor & Francis,*, vol. 17, no. 4, pp. 648–674, 2021.

[301] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe, "Weisfeiler and leman go neural: Higher-order graph neural networks," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, pp. 4602–4609, 2019.

[302] K. K. Thekumparampil, C. Wang, S. Oh, and L.-J. Li, "Attention-based graph neural network for semi-supervised learning," *arXiv preprint arXiv:1803.03735*, 2018.

[303] D. Mesquita, A. Souza, and S. Kaski, "Rethinking pooling in graph neural networks," *Advances in Neural Information Processing Systems, ACM New York, NY*, vol. 33, pp. 2220–2231, 2020.

[304] P. J. M. Ali, R. H. Faraj, E. Koya, P. J. M. Ali, and R. H. Faraj, "Data normalization and standardization: a technical report," *Machine Learning Technical Reports*, vol. 1, no. 1, pp. 1–6, 2014.

[305] Z. Aghababaeyan, M. Abdellatif, M. Dadkhah, and L. Briand, "Deepgd: A multi-objective black-box test selection approach for deep neural networks," *arXiv preprint arXiv:2303.04878*, 2023.

[306] A. Zolfagharian, M. Abdellatif, L. C. Briand, M. Bagherzadeh, and S. Ramesh, "A search-based testing approach for deep reinforcement learning agents," *IEEE Transactions on Software Engineering, IEEE*, 2023.

[307] Z. Aghababaeyan, M. Abdellatif, L. Briand, S. Ramesh, and M. Bagherzadeh, "Black-box testing of deep neural networks through test case diversity," *IEEE Transactions on Software Engineering, IEEE*, 2023.

[308] Z. Yu, N. A. Kraft, and T. Menzies, "Finding better active learners for faster literature reviews," *Empirical Software Engineering, Springer,*, vol. 23, pp. 3161–3186, 2018.

[309] J. Chen, A. Schein, L. Ungar, and M. Palmer, "An empirical study of the behavior of active learning for word sense disambiguation," in *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pp. 120–127, ACM New York, NY, 2006.

[310] M. Sassano, "An empirical study of active learning with support vector machines forjapanese word segmentation," in *Proceedings of the 40th annual meeting of the association for computational linguistics*, pp. 505–512, Association for Computational Linguistics, 2002.

[311] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, pp. 681–694, 2020.

[312] S. Alharbi, M. Alrazgan, A. Alrashed, T. Alnomasi, R. Almojel, R. Alharbi, S. Alharbi, S. Alturki, F. Alshehri, and M. Almojil, "Automatic speech recognition: Systematic literature review," *Ieee Access*, vol. 9, pp. 131858–131876, 2021.