PhD-FSTM-2024-072
Faculty of Science, Technology and Medicine

Dissertation

Presented on the 16/09/2024 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Yinghua LI
Born on 28th June 1992 in Sichuan, China

Test Input Prioritization for Deep Neural Networks

Dissertation Defense Committee

Dr. Tegawendé F. BISSYANDE, Dissertation Supervisor
Professor, University of Luxembourg, Luxembourg

Dr. Jacques KLEIN, Chairman
Professor, University of Luxembourg, Luxembourg

Dr. Lei MA, Vice Chairman
Professor, The University of Tokyo, Japan

Dr. Hongyu ZHANG, Member
Professor, Chongqing University, China

Dr. Xiang GAO, Member
Professor, Beihang University, China

Abstract

The rapid adoption of deep neural networks (DNNs) has revolutionized machine learning in several domains. As a result, thorough evaluation and validation of DNNs are crucial for ensuring their effectiveness. Testing DNNs, however, is challenging due to three key issues: 1) manual labeling is the mainstream; 2) test sets can be large scale; and 3) domain-specific knowledge can be required for labeling. To reduce the labeling costs, one promising approach is test prioritization, which focuses on identifying and prioritizing potentially misclassified test inputs. Early identification of such challenging inputs can accelerate the DNN debugging process and improve the efficiency of DNN testing. While existing test prioritization approaches for DNNs have proven effective in some cases, they show limitations when applied to more specialized scenarios.

In this dissertation, we focus on four special scenarios, namely video classification, Graph Neural Networks (GNN) classification, compressed DNN classification and 3D shape classification. Applying existing DNN test prioritization methods to these four specific domains presents certain limitations. From Chapter 3 to Chapter 6, each chapter proposes a new test prioritization method aimed at a specific domain. We conducted empirical studies to demonstrate their effectiveness. Below, we present the core contributions.

- **Test prioritization for videos** To solve the labeling-cost problem specifically in the context of video test inputs, we proposed a novel test prioritization approach called VRank. The fundamental concept underlying VRank is that test inputs situated closer to the decision boundary of the model are at a higher risk of being predicted incorrectly. To capture the spatial relationship between a video test and the decision boundary, we designed a series of feature generation strategies tailored to video-type tests. Based on these strategies, VRank generated features for each test in the test set to perform test prioritization.
- **Test prioritization for GNNs** To relieve the labeling-cost problem and improve the efficiency of GNN testing, we propose a GNN-oriented test prioritization approach, NodeRank. NodeRank leverages the concepts of mutation testing to perform test prioritization, operating on the core premise that if a test input (node) can kill many mutated models and produce different prediction results with many mutated inputs, this input is considered more likely to be misclassified by the GNN model and should be prioritized higher.
- **Test prioritization for compressed DNNs** To address the challenge of labeling-cost reduction in testing compressed DNN models, we proposed PriCod, which can identify and prioritize potentially misclassified tests. PriCod leverages the behavior disparities caused by model compression, along with the embeddings of test inputs, to effectively prioritize potentially misclassified tests.
- **Test prioritization for 3D point clouds** To address the issue of high labeling

costs for 3D point cloud data, we propose a novel test prioritization approach, PCPrior. PCPrior relies on the premise that test inputs closer to the decision boundary of the model are more likely to be predicted incorrectly. To this end, we designed a set of feature generation strategies tailored to 3D point clouds and utilized the generated features for test prioritization.

In summary, this dissertation proposes four new test prioritization methods tailored to four specialized DNN scenarios and demonstrates their effectiveness against the compared methods.

Well done is better than well said.

Benjamin Franklin

Acknowledgements

I would like to express my deepest gratitude to those who supported and helped me throughout my PhD journey. Their support was indispensable, and without their assistance, the completion of my dissertation would not have been possible. It is my pleasure to express my gratitude to them.

Firstly, I wish to express my heartfelt gratitude to my supervisor, Prof. Tegawendé F. Bissyandé. He has always trusted and supported me with his great kindness throughout my whole PhD journey. I am particularly grateful for his patient communication and continuous encouragement in my research. His academic diligence and persistence had a profound positive impact on my personal growth and development, inspiring me to continue engaging in academic research in my future career.

Next, I am equally grateful to my daily advisor, Prof. Jacques Klein. He was always willing to provide valuable suggestions and in-depth discussions for my research work. I extend profound thanks to my co-supervisor, Prof. Lei Ma, for his patience, advice, guidance, and encouragement in my research. Special thanks to Doctoral Researcher Xueqi Dang, who regularly discussed research project with me and persistently offered help and support throughout my PhD journey.

Thirdly, I would like to thank all my co-authors who worked closely with me during my PhD study for their helpful discussions and collaboration. Specifically, I am grateful to Dr. Jun Gao for helping me get acquainted with the HPC computing cluster, ensuring all my experiments could run smoothly on HPC during my PhD. Moreover, I would like to express my gratitude to the HPC teams for providing robust computing resources for my research.

I would like to thank all the members of my PhD defense committee, including Prof. Jacques KLEIN, Prof. Lei Ma, Prof. Hongyu Zhang, Prof. Xiang Gao and my supervisor Prof. Tegawendé F. Bissyandé. It is my great honor to have them participate in my defense committee, and I am very grateful for their review of my dissertation and my PhD work.

I would like to thank all my colleagues from TruX (SnT) for all the good discussions and interesting reading group sessions.

Finally, I express my deepest gratitude to my parents. They supported all my decisions unconditionally and constantly encouraged me to move forward and pursue my dreams.

Yinghua Li
University of Luxembourg
August 2024

Contents

List of Figures

List of Tables

1 Introduction

In this chapter, we first introduce the motivation of DNN test prioritization. Then, we describe the limitations of existing DNN test prioritization methods. Finally, we present the contributions and the roadmap of this dissertation.

Contents

1.1 Motivation

The widespread adoption of deep neural networks (DNNs) has led to significant advancements in machine learning, particularly in domains such as computer vision [1, 2], autonomous vehicles [3, 4], and recommendation systems [5]. Consequently, the evaluation and validation of DNNs have become increasingly critical to ensure their performance. However, as mentioned in the existing study [6], testing DNNs poses significant challenges. This challenge stems from three main factors: 1) manual labeling is still the mainstream, typically necessitating the involvement of multiple annotators to ensure the accuracy and consistency of the labeling process.; 2) test sets can be large and complex 3) labeling can require domain-specific knowledge, which can be costly to acquire.

To minimize labeling costs and enhance the efficiency of testing DNN-based systems, an intuitive approach is to prioritize test inputs that are more likely to be misclassified. By prioritizing these inputs, resources can be allocated to label these potentially misclassified inputs first. Early identification of these inputs can rapid the DNN debugging process, significantly improving the efficiency of DNN testing.

In the literature, numerous test prioritization strategies have been proposed for DNNs, broadly classified into three categories: coverage-based [7, 8], confidence-based [6, 9], and mutation-based [10] methods. Coverage-based techniques, such as CTM [11], prioritize test inputs by considering neuron coverage, borrowing concepts from traditional software testing [11, 12]. Confidence-based approaches [6, 10] operate under the assumption that test inputs with lower model confidence are more likely to be misclassified and should be prioritized higher. Prior work [6] has demonstrated that confidence-based approaches perform better than coverage-based test prioritization approaches. Mutation-based test prioritization methods focused on employing mutation operations and supervised learning for prioritizing tests. PRIMA [10], a mutation-based strategy, utilizes well-designed model mutation rules and input mutation rules to generate mutation results and employs the ranking model [13] for efficient test input prioritization.

Although the aforementioned approaches have been demonstrated to be effective in some cases, they exhibit limitations in some specialized scenarios. In our work, we concentrate on four special scenarios: three-dimensional (3D) shape classification, Graph Neural Networks (GNN) classification, compressed DNN classification, and video classification. From Chapter 3 to Chapter 6, each chapter proposed a new test prioritization method aimed at a specific domain. The core motivation behind all the methods is to address the labeling cost problem. Specifically, these proposed approaches prioritize test inputs that are more likely to be misclassified by the model. Early identification and labeling of such inputs can reduce manual labeling efforts and improve the overall efficiency of the testing process.

Moreover, we conducted empirical studies to demonstrate that our proposed new methods outperform existing test prioritization approaches. Below, we provide detailed explanations of our focused scenarios.

- **Video classification** Video classification focuses on categorizing input videos into predetermined classes. With the growing prominence of short-form videos, there is an increased demand for efficient video classification algorithms [14], and the presence of bugs in video-oriented DNNs can lead to substantial real-world consequences. For instance, consider a scenario on a highway where a camera-

equipped model aims to detect car accidents. If the model inaccurately classifies an accident scene as safe, it will fail to issue a timely warning, potentially leading to severe consequences due to delayed assistance. Hence, ensuring the quality of video classification models is crucial.

- **GNN classification** GNNs have emerged as powerful tools for capturing intricate relationships within graph-structured data. In GNNs, a graph is typically composed of nodes and edges. For example, in the case of the Cora dataset [15], given a test input (a scientific paper), a GNN model can be used to classify the node into specific categories (e.g., 'neural networks paper'). With the application of GNNs continuing to expand, the testing and validation of GNNs becomes increasingly essential.
- **Compressed DNN classification** Compressed DNN models are engineered to minimize computational and memory requirements while preserving performance, thereby enabling effective deployment in resource-constrained contexts. Various compression techniques have proven to be valuable in reducing the size and computational load of DNNs while maintaining their predictive capabilities. Consequently, the evaluation and validation of compressed DNNs have become increasingly critical to ensure their performance on devices.
- **3D shape classification** 3D shape classification refers to the task of categorizing three-dimensional shapes [16]. In this context, the input consists of the geometric information of three-dimensional objects, typically represented as point clouds. A point cloud [17] consists of a collection of three-dimensional data points in space. Compared to two-dimensional data (e.g., images), 3D point clouds can provide a three-dimensional depiction of objects, thereby enhancing accuracy and reliability in identifying complex 3D shapes and volumes. Moreover, point cloud data can directly capture surface details and morphology of objects. Consequently, the integration of point cloud processing in safety-critical applications, such as autonomous driving [18, 4] has become increasingly prevalent. In recent years, the application of DNNs to 3D point cloud data has garnered significant attention. Ensuring the reliability of DNNs operating on 3D point cloud data is crucial for safe and efficient functioning.

Our work focuses on proposing new test prioritization methods for the aforementioned four scenarios in order to accelerate the debugging process and enhance the overall testing efficiency in these contexts. In the following section, we will discuss the limitations of applying existing test prioritization methods to the aforementioned scenarios.

1.2 Limitations of Existing Approaches

In this section, we discuss the limitations of existing test prioritization methods when applied to the four specific scenarios, namely video classification, GNN classification, compressed DNN classification, and 3D shape classification.

1.2.1 Video classification

When applying confidence-based approaches to video-type test inputs, the following limitations occur: they do not take into account the unique temporal information present in video data. In contrast to images and text, video inputs consist of multiple frames that capture the dynamic nature and temporal fluctuations of objects over time. On the other hand, the mutation-based test prioritization approach PRIMA is

not applicable to video test inputs because the mutation rules of PRIMA are not adapted for video datasets.

1.2.2 GNN classification

When applying confidence-based approaches to GNNs, they have the following limitations:

- Confidence-based approaches do not account for the interdependence present in graph-structured test inputs. Specifically, graph inputs are composed of nodes interconnected by edges, and this interdependence plays a crucial role in the inference process of GNNs. However, confidence-based approaches were originally designed for DNNs, whose tests are typically independent of each other. They ignore the interdependence in the graph-structured data in the process of test prioritization.
- Confidence-based approaches operate under the assumption that test inputs for which the model exhibits low confidence are more likely to be misclassified and, therefore, should be prioritized higher. However, in the presence of adversarial attacks, the model's confidence can be higher for incorrect predictions. In such cases, even if the model is highly confident in its prediction for a test, it does not necessarily imply that the test is more likely to be misclassified.

The mutation-based method, PRIMA, cannot be applied to GNNs since their mutation operators are not adapted to graph-structured data and GNN models.

1.2.3 Compressed DNN classification

When applying confidence-based approaches to compressed DNN models, the following limitations occur: they treat the compressed DNN models as black boxes and ignore the information regarding deviations before and after model compression when conducting test prioritization. Moreover, the mutation-based test prioritization approach, PRIMA [10], cannot be applied to compressed DNN models because the model mutation operators of PRIMA are not applicable to them. This limitation arises from the fact that the architectures and gradients of compressed models are typically unavailable [19].

1.2.4 3D shape classification

When applying confidence-based test prioritization approaches to 3D point cloud data, there are the following limitations.

- **Noises in 3D point cloud data** 3D point cloud data typically contains noises (e.g., sensor noise and non-uniform sampling density). These noises can reduce the effectiveness of confidence-based approaches. Specifically, for a given test sample, the model can erroneously assign a high probability to an incorrect category due to the noise. In this case, confidence-based approaches will assume that the model is highly confident of this particular test, considering it will not be misclassified. However, the model's prediction on this test sample is indeed incorrect.
- **Missing crucial spatial features** Confidence-based methods operate solely based on the model's prediction uncertainty on tests. However, 3D point cloud typically exhibits complex spatial characteristics, and confidence-based methods fail to fully leverage the informative features inherent in point cloud data for test prioritization.

When applying the mutation-based test prioritization method to 3D point clouds,

there are the following limitations.

- The mutation operators utilized in PRIMA are primarily designed for two-dimensional data. These operators are not directly applicable to 3D point cloud data. Unlike traditional image or text data, 3D point clouds have a unique three-dimensional representation characterized by a large number of points.
- Even considering the possibility of converting 3D data into 2D images using dimensionality reduction techniques and integrating them into PRIMA, there are some practical problems. Specifically, PRIMA requires feeding 2D images of mutations into the DNN model to compare predictions between the mutants and the original input. However, models designed for 3D point clouds are inherently tailored to process three-dimensional data and lack the capability to classify two-dimensional images. Therefore, these models cannot make predictions for the variants. Consequently, even in scenarios where dimensionality reduction tools are available, PRIMA remains unsuitable for accommodating 3D point cloud data.

1.3 The Main Contributions

In this section, we provide a summary of the contributions of this dissertation.

- **Prioritizing Test Cases for Deep Learning-based Video Classifiers** We propose VRank, a novel test prioritization approach designed specifically for video test inputs. The key premise is that test inputs situated closer to the decision boundary of the model are at a higher risk of being predicted incorrectly. To capture the spatial relationship between a video test and the decision boundary, we employ a vectorization technique that transforms a given video test into a lower-dimensional space to indirectly reveal the underlying proximity between the test and the decision boundary. To implement this vectorization strategy, we generate four different types of features for each video-type test: temporal features, video embedding features, prediction features, and uncertainty features. By combining these feature types, VRank effectively constructs a comprehensive feature vector for each individual test input. To assess the misclassification likelihood of each test input, VRank employs a LightGBM-based ranking model that takes the constructed feature vector as input and generates a misclassification score. Based on these misclassification scores, VRank sorts all the tests within the test set in descending order. We conducted an empirical evaluation to assess the performance of VRank, encompassing both natural and noisy datasets. The experimental results demonstrated that VRank outperforms all the compared test prioritization methods.

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

- **Test Input Prioritization for Graph Neural Networks** We propose NodeRank, a novel test input prioritization approach targeting GNNs. The core idea is that a test is considered more likely to be misclassified if it can kill many mutated models and produce different prediction results with many mutated inputs. To this end, we developed three different types of mutations, namely graph structure mutation, node feature mutation, and GNN model mutation, based on the characteristics of GNNs and the graph test dataset. For each test input, NodeRank generates these three types of mutations. By comparing the prediction results before and after the mutation, NodeRank generates a mutation feature vector for

each test. We trained an ensemble ranking model to predict the misclassification score for each test based on its mutation feature vector and accordingly ranked all the tests. We conducted an empirical study to evaluate the effectiveness of NodeRank. The results demonstrated that NodeRank outperformed all the compared test prioritization approaches.

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

- **PriCod: Prioritizing Test Inputs for Compressed Deep Neural Networks** We propose PriCod, a novel test prioritization approach designed for compressed DNNs. PriCod leverages the behavior disparities caused by model compression, along with the embeddings of test inputs, to effectively prioritize potentially misclassified tests. It operates on the premise that significant behavior disparities between the models indicate potential misclassifications and that inputs near decision boundaries are more likely to be misclassified. To this end, PriCod generates deviation features and embedding features for each test input to capture the prediction deviation caused by model compression and the proximity to decision boundaries, respectively. By combining these features, PriCod predicts the probability of misclassification for each test, ranking tests accordingly. We conduct an extensive study to evaluate the effectiveness of PriCod on natural, noisy, and adversarial test inputs. The experimental results demonstrate that PriCod outperforms all the compared test prioritization approaches in all three types of scenarios.

  This work has undergone major revision and is currently under review in ACM Transactions on Software Engineering and Methodology(TOSEM) in 2024.

- **Test Input Prioritization for 3D Point Clouds** We proposed PCPrior, a novel test prioritization approach specifically designed for 3D point cloud test cases. PCPrior leverages the unique characteristics of 3D point clouds to prioritize tests. The core idea is that test inputs closer to the decision boundary of the model are more likely to be predicted incorrectly [20]. To capture the spatial relationship between a point cloud test and the decision boundary, we propose transforming each test (a point cloud) into a low-dimensional feature vector, towards indirectly revealing the underlying proximity between a test and the decision boundary. To achieve this, we carefully design a group of feature generation strategies, and for each test input, we generate four distinct types of features, namely, spatial features, mutation features, prediction features, and uncertainty features. Through a concatenation of the four feature types, PCPrior assembles a final feature vector for each test. Subsequently, PCPrior employs a ranking model to estimate the misclassification probability for each test based on its feature vector. Finally, PCPrior ranks all tests based on their misclassification probabilities. We conducted an extensive study based on 165 subjects to evaluate the performance of PCPrior, encompassing both natural and noisy datasets. The results demonstrate that PCPrior outperforms all the compared test prioritization approaches.

  This work has been accepted by the ACM Transactions on Software Engineering

and Methodology (TOSEM) in 2024.

1.4 Roadmap

The dissertation roadmap is depicted in Figure 1.1. Chapter 2 provides the background and related work concerning deep neural networks, test input prioritization for DNNs, DNN testing, and mutation testing. Chapter 3 presents our proposed test prioritization approach designed for video-type test inputs, called VRank. Chapter 4 presents our proposed test prioritization approach specifically designed for GNNs, called NodeRank. Chapter 5 presents our proposed test prioritization approach tailored for compressed DNN models, PriCod. Chapter 6 presents our proposed test prioritization approach tailored to 3D point clouds, PCPrior. Chapter 7 concludes the dissertation and discusses future work.

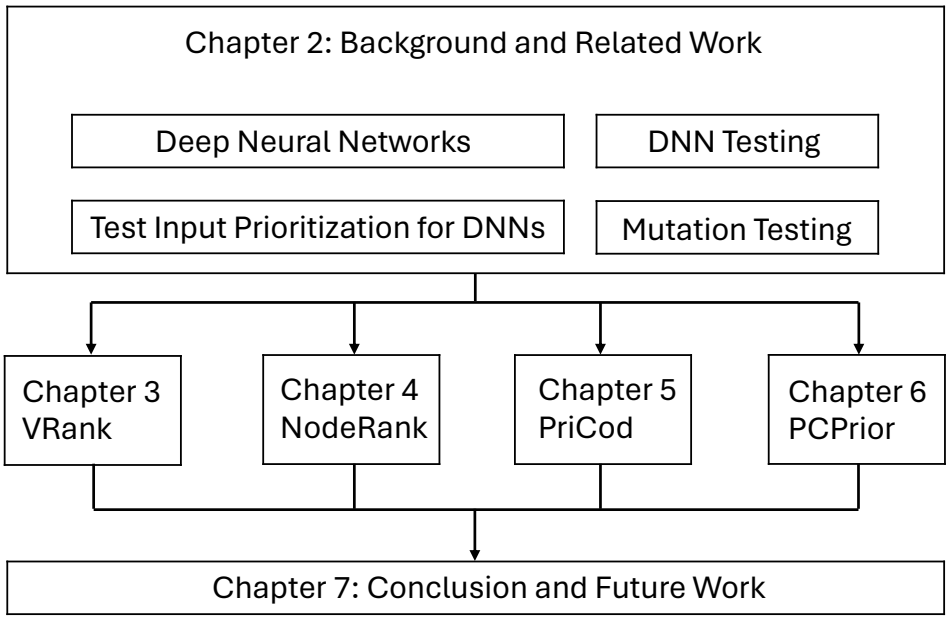


Figure 1.1: Roadmap of this dissertation

2 Background and Related Work

This chapter introduces the necessary concepts and related works to understand this dissertation. We present the details of deep neural networks, test input prioritization for DNNs, DNN testing and mutation testing, respectively.

Contents

2.1 Deep Neural Networks Across Diverse Domains

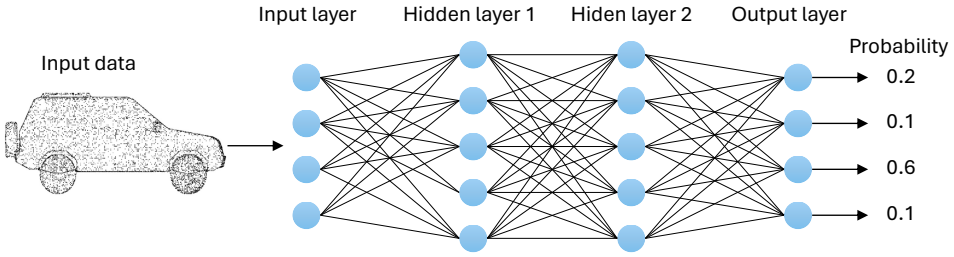


Figure 2.1: An example of a DNN classifier

Deep neural networks (DNNs) serve as the core of numerous deep learning applications, particularly in tasks related to classification. As illustrated in Figure 2.1, the architecture of a DNN consists of multiple layers: the input layer, a number of hidden layers (which are optional), and the output layer. Each layer consists of a series of neurons, which are the fundamental processing units in a DNN. Neurons from different layers are interconnected through weighted edges. These neurons gather and process information from the input or previous neurons, utilizing activation functions to produce outputs. The weights of the connections between neurons are not predetermined by developers. Instead, they are adjusted during a training phase that employs a large dataset of labeled data.

Once sufficiently trained, a DNN can autonomously classify inputs. For instance, as shown in Figure 2.1, a DNN is used to determine a car's type from its image. For any given input, this DNN generates an N-dimensional vector, denoted as $\langle p_1, p_2, \ldots, p_N \rangle$, where each p_i refers to the likelihood that the input image corresponds to the i-th category. The sum of all p_i values equals 1. In Figure 2.1, the DNN predicts the probabilities of the input belonging to each category as 0.2, 0.1, 0.6, and 0.1, respectively.

In the following, we introduce some specific scenarios of DNNs, including video classification, graph neural networks, model compression, and 3D point cloud processing. From Chapter 3 to Chapter 6, the test prioritization methods we propose are specifically designed for these scenarios.

2.1.1 DNNs for Video Classification

In recent years, the proliferation of multimedia content on the Internet has surged, leading to a significant increase in the volume of videos shared every minute. This rapid expansion highlights the need for comprehensive analysis of video data. In the field of computer vision [21], researchers have dedicated efforts to developing algorithms to address video analysis challenges, particularly video classification. Tran *et al.* [22] proposed the use of deep 3D ConvNets to extract spatio-temporal features for video classification, leveraging extensive video datasets. Building upon this work, Tran *et al.* [23] conducted an empirical ConvNet architecture search to improve spatiotemporal feature learning, which outperformed C3D on several datasets, with faster inference time, smaller model size, and more compact representation. Additionally, Feichtenhofer *et al.* [24] introduced the SlowFast network for video recognition, incorporating a Slow pathway for capturing spatial semantics and a Fast pathway for discerning motion at a finer temporal resolution.

2.1.2 Graph Neural Networks

Graph Neural Networks (GNNs) have demonstrated remarkable effectiveness in addressing machine learning challenges associated with data structured in graphs [25, 26, 27]. Within the domain of GNNs, a graph is defined by nodes and edges, typically denoted as $G = (V, E)$, where V represents the set of nodes and E denotes the connections between them. One practical application of GNNs is node classification. A prevalent node classification dataset is Cora, where nodes correspond to scientific papers, and edges represent citations between them. Given an input (a scientific paper), a GNN model can classify the paper into specific categories such as 'reinforcement learning' and 'neural networks'.

[**GNN training**] GNNs undergo a training process similar to other neural networks. The necessary inputs for GNN training generally consist of: **1) Graph Structure** This encompasses the connections between nodes in the graph. **2) Node Features** Each node typically possesses associated feature vectors, reflecting its attributes. **3) Target Labels** In GNN node classification training data, 'Target Labels' denote the category to which each node belongs, usually predefined.

[**GNN inference**] In the context of GNNs, inference denotes utilizing a pre-trained GNN model to make predictions on new graph data. For instance, in node classification tasks, the GNN model leverages its learned parameters and weights to classify a given node. The input typically encompasses the features of the node and the graph structure to which it belongs. The output is the classification of the node.

2.1.3 DNN Model Compression

DNN model compression has emerged as a critical research focus, tackling the challenge of deploying DNN models in environments constrained by computational and storage resources [28, 29, 30]. The aim of DNN model compression is to reduce the size and computational demands of DNN models without significantly reducing their performance, thereby enabling their deployment on mobile devices like smartphones. Among various compression techniques, quantization stands out as a prevalent method. Model quantization aims to reduce the precision of the model's numerical parameters, typically converting 32-bit floating-point weights to lower-bit representations such as 8-bit, thereby decreasing both the model's storage requirements and computational complexity [31]. TensorFlow Lite (TFLite) [32] and CoreML [33] are two widely-adopted model quantization approaches. TFLite, developed by Google, specializes in optimizing neural networks for mobile and embedded device deployment, with a focus on efficient computational performance on Android devices [34]. On the other hand, CoreML is Apple's framework tailored for iOS devices, leveraging Apple hardware to enable rapid neural network inference through hardware acceleration.

2.1.4 Deep Learning for 3D Point Clouds

The emergence of advanced sensor technologies, such as LiDAR (Light Detection and Ranging) [35] and RGB-D (Red-Green-Blue Depth) cameras [36], has led to an explosion of 3D point cloud data across various applications. A 3D point cloud typically represents a collection of data points in 3D space [37]. The emergence of Deep Learning [38], has revolutionized the analysis and understanding of 3D point cloud data. Moreover, the accessibility of openly accessible datasets, such as ModelNet [39], ShapeNet [40], and S3DIS [41], has played a pivotal role in advancing

research on the application of deep learning techniques to 3D point clouds. One crucial aspect of research in this domain is 3D shape classification, which focuses on leveraging DNNs to classify three-dimensional shapes. For instance, in autonomous driving applications, 3D shape classification can be used to categorize objects on the road, such as vehicles and traffic signs. Accurate classification of these objects can enhance the autonomous driving system's understanding of its surroundings, leading to more precise decision-making. In the literature [17, 16, 38, 42], several approaches have been proposed to tackle the challenge of 3D shape classification. Notable methods include PointConv [17], Dynamic Graph Convolutional Neural Network (DGCNN) [16], and PointNet [38]. PointConv, a specialized convolutional neural network, is tailored for the processing of 3D point clouds. By training multi-layer perceptrons on local point coordinates, it facilitates the direct construction of deep networks on 3D point clouds, enabling efficient analysis. DGCNN, on the other hand, treats 3D point cloud data as graphs and exploits intrinsic spatial relationships. Utilizing graph convolutions and dynamically adapting the graph structure based on input data, DGCNN effectively learns and processes point cloud representations. PointNet, another widely adopted architecture for processing 3D point cloud data, integrates a shared multi-layer perceptron (MLP) with max-pooling for local feature extraction, along with a symmetric function for aggregating global features. T-Net layers within PointNet enable the learning of transformation matrices, thereby enhancing its robustness to input variations.

2.2 Test Input Prioritization for DNNs

Test prioritization is a critical aspect of software testing, aiming to determine the optimal sequence for conducting unlabeled tests [6]. Its primary objective is to identify and prioritize tests that are likely to be misclassified, thereby facilitating early labeling and improving debugging efficiency. Current test prioritization strategy for Deep Neural Networks can be broadly classified into three categories: coverage-based [7, 8], confidence-based [6, 9], and mutation-based [10] methods.

Coverage-based techniques, such as CTM [11], prioritize test inputs based on neuron coverage and adapt coverage-based prioritization techniques from traditional software testing [11, 12]. Pei *et al.* proposed the basic neuron coverage criterion inspired by program coverage. These metrics can be used as test prioritization metrics. Ma *et al.* proposed DeepGauge [7], which defines several coverage-based criteria that can be used for test prioritization, such as KMNC and Neuron Boundary Coverage. Confidence-based approaches [6, 10] assume that test inputs with lower model confidence are more likely to be misclassified and, hence, should be prioritized higher. Existing studies [6] have shown that confidence-based approaches are more effective than coverage-based ones. DeepGini [6] is a classical confidence-based test prioritization approach, which assumes that a test input is more likely to be mispredicted if the DNN outputs similar probabilities for each class. Weiss *et al.* [9] performed a comprehensive investigation of various DNN test input prioritization techniques, including several confidence-based approaches such as Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy. These demonstrated that these simple confidence-based approaches are effective in identifying possibly misclassified test inputs. Mutation-based test prioritization methods focus on utilizing mutation operations and supervised learning for test prioritization. PRIMA [10], is a mutation-based test prioritization strategy, which employs well-designed model and input

mutation rules to generate mutation results and leverages the XGBoost ranking algorithm [13] to achieve effective test input prioritization.

In addition to the aforementioned work, Zheng *et al.* [43] proposed CertPri, which prioritizes test inputs by measuring movement difficulty in the feature space. Specifically, to perform test prioritization, CertPri evaluates the cost of moving test inputs closer to or farther from the class centers. Wei *et al.* [44] proposed EffiMAP, an efficient test prioritization technique leveraging predictive mutation analysis. Instead of requiring a complete mutation analysis, EffiMAP predicts the ability of test cases to expose model prediction failures by utilizing information extracted from the test case execution trace. Tao *et al.* [45] proposed TPFL, which employs dynamic spectrum analysis at the neuron level for test prioritization. TPFL identifies neurons that likely cause incorrect decisions in a DNN and ranks test inputs based on their potential to activate these suspicious neurons.

2.3 DNN Testing

To optimize the efficiency of DNN testing, numerous test optimization approaches have been introduced in the existing literature [6, 46, 10, 47]. These approaches typically fall into two main categories: test input prioritization [6, 9] (which has been discussed in the previous section) and test selection [46, 48], which aims to select a small subset of test inputs to accurately estimate the overall accuracy of the test set, thereby reducing labeling costs. Li *et al.* [48] proposed Cross Entropy-based Sampling (CES) to select representative test inputs for DNN accuracy estimation. CES operates by minimizing the cross-entropy between the selected set and the entire test set, thereby ensuring similarity in distribution. Chen *et al.* [46] proposed PACE for accuracy estimation. PACE first clusters all the test inputs and then employs the MMD-critic algorithm [49] to select prototypes from each cluster. Finally, all the selected tests are aggregated for accuracy estimation. Zhou *et al.* [50] proposed DeepReduce to estimate the performance of a DL model. DeepReduce begins by selecting a subset of test data to satisfy testing adequacy. It then selects additional data to approximate the output distribution of the entire testing dataset, using relative entropy minimization to minimize the difference.

In addition to focusing on DNN testing efficiency, several studies [51, 20, 8, 7, 52] have concentrated on evaluating DNN testing adequacy. Pei *et al.* [8] introduced neuron coverage to gauge the coverage of DNN model logic by a test set. Ma *et al.* [7] proposed DeepGauge, a suite of coverage-based metrics that regard neuron coverage as a crucial indicator of test input effectiveness. Moreover, they devised metrics based on neuron coverage granularity to distinguish adversarial attacks from legitimate test data. Kim *et al.* [52] proposed surprise adequacy, which focused on assessing the efficacy of a test input within a test set by measuring its surprise concerning the training set. Surprise is defined as the disparity in neuron activation values when presented with this new input. Beyond these, Dola *et al.* [53] proposed the Input Distribution Coverage (IDC) framework for evaluating the black-box test adequacy of DNNs. Leveraging a Variational Autoencoder (VAE) to transform test inputs into feature vectors, IDC establishes a coverage domain wherein Combinatorial Interaction Testing (CIT) metrics measure test coverage.

2.4 Mutation Testing

Mutation testing is a systematic software testing technology that has attracted great attention from academia and industrial research circles [54]. The basic principle consists of introducing small and intentional modifications (called mutants) into the source code of a software system to simulate potential failures that may occur during program execution. A well-designed test suite should be able to detect these mutants, demonstrating its effectiveness in identifying real bugs in the code [55]. In traditional software testing [56, 57, 58], mutation testing can assess the fault-detection capabilities of individual test cases, facilitating test prioritization. Lou *et al.* [56] proposed a novel test-case prioritization approach that orders test cases based on their fault detection ability, determined through the analysis of mutation faults simulated from real software faults. By strategically ordering test cases, this approach aims to maximize testing efficiency by prioritizing the detection of critical faults. Shin *et al.* [57] proposed a method that combines mutation-based and diversity-based approaches for test case prioritization, demonstrating the effectiveness of mutation-based prioritization in comparison to random and coverage-based prioritization.

Furthermore, in addition to the context of traditional software, numerous studies have explored the application of mutation testing to DNNs. Ma *et al.* [20] proposed DeepMutation, which utilized mutation testing to evaluate the quality of DNN test data. They designed a collection of mutation operators to inject faults into training data, programs, and DL models. DeepMutation evaluated the validity of the test data by analyzing the extent to which injected faults can be detected. Hu *et al.* [59] extended the work of Ma *et al.* [20], and thereby developed DeepMutation++, which introduced new mutation operators for feedforward neural networks (FNN) and recurrent neural networks (RNN) and enabled mutation of the RNN runtime state. Humbatova *et al.* [60] proposed DeepCrime, a mutation testing tool that implemented a set of DL mutation operators based on real DL faults. Shen *et al.* [61] proposed MuNN, a method for mutation analysis specific to neural networks. MuNN defines five mutation operators grounded in neural network characteristics.

3 Prioritizing Test Cases for Deep Learning-based Video Classifiers

In this chapter, we propose a novel test prioritization approach called VRank. VRank is designed to assign higher priority to video test inputs that are more likely to be misclassified. The fundamental concept is that test inputs situated closer to the decision boundary of the model are at a higher risk of being predicted incorrectly. By identifying and prioritizing such inputs, developers can allocate limited label budgets to such potentially misclassified inputs and improve testing efficiency.

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

- Yinghua Li, Xueqi Dang, Lei Ma, Jacques Klein and Tegawendé F. Bissyandé. Prioritizing Test Cases for Deep Learning-based Video Classifiers. Empirical Software Engineering (EMSE). Accepted for publication on Jun. 20, 2024.

Contents

3.1 Introduction

The rapid growth of multimedia on the Internet has led to an exponential increase in the number of videos being shared every minute. The popularity of short videos has further heightened the demand for video classification algorithms [14] to facilitate speedy user video recommendations [62]. Specifically, video classification plays a crucial role in identifying and tracking objects in a variety of domains, such as accident detection [63, 64, 65]. Given the crucial usage, the presence of bugs in video-oriented Deep Neural Networks (DNNs) can have severe real-world consequences, especially in safety-critical domains [66]. Here, bugs refer to certain internal parameter weights within the video classification model that can lead to prediction errors when dealing with video inputs. For example, consider a highway scenario where the camera-equipped video classification model is specifically engineered to determine whether a given scene involves a car accident. In the event of an erroneous prediction, where the model misclassified the accident scene as a safe scenario, there is a risk of failing to issue a timely warning. This oversight can potentially result in severe consequences due to a lack of prompt assistance. Therefore, it is crucial to guarantee the quality of DNN models employed for video classification.

DNN testing [67] is widely recognized as an effective means of ensuring the quality of DNNs, including DNNs for video classification. However, a significant challenge in DNN testing lies in the high cost associated with labelling test inputs to verify the accuracy of DNN predictions. The general reasons include: 1) the test set is usually large-scale; 2) manual labelling is still mainstream; 3) labelling can require domain-specific expertise. Furthermore, in comparison to labeling image and text data, labeling video-type test inputs presents unique challenges, outlined as follows.

- Video data is characterized by its sequential composition of frames, establishing a temporal structure. Unlike static images or text, video data necessitates annotators to meticulously observe and analyze the content over time, frame by frame.
- Video datasets can contain multiple objects and events within a single frame, making it challenging to identify which objects or events should be labelled.
- Video datasets are typically much larger than image/text datasets, containing multiple frames per second, which can create a large volume of data to be labelled. This can be time-consuming and resource-intensive, requiring significant human labor.

To relieve the labelling cost problem, one effective way is test prioritization [6], which aims to prioritize bug-revealing test inputs (i.e., test inputs that are more likely to be misclassified by the DNN model) earlier in the testing process so that those test inputs can be labeled earlier. To this end, researchers have proposed several test input prioritization techniques to address the labelling-cost issue in DNNs [10, 6]. These techniques can be broadly categorized into coverage-based and confidence-based approaches. Coverage-based approaches, such as CTM [11], prioritize test inputs based on neuron coverage and adapt coverage-based prioritization techniques from traditional software testing [11, 12]. On the other hand, confidence-based approaches [6, 10] assume that test inputs with lower model confidence are more likely to be misclassified and hence should be prioritized higher. DeepGini [6], a classical confidence-based test prioritization approach, considers a test input more likely to be misclassified if the model outputs similar prediction probabilities for each class. Wang *et al.* [10] proposed PRIMA, which leverages mutation analysis

and learning-to-rank methods to prioritize test inputs for DNNs.

However, when applying the aforementioned existing test prioritization methods to the scenario of video test inputs, certain limitations arise:

- The approaches mentioned above do not take into account the unique temporal information present in video data. In contrast to images and text, video inputs consist of multiple frames that capture the dynamic nature and temporal fluctuations of objects over time.
- The mutation-based test prioritization approach PRIMA is not applicable to video test inputs because the mutation rules of PRIMA are not adapted for video datasets.

In this paper, we propose VRank (**V**ideo Test Inputs **Rank**ing), the first test input prioritization technique tailored exclusively for video test inputs. The fundamental concept underlying VRank is that video-type tests with a higher probability of being misclassified by the evaluated DNN classifier are considered more likely to reveal faults and will be prioritized higher. To achieve this, we train a ranking model with the goal of predicting the probability of a given test input being misclassified by a DNN classifier. Specifically, the ranking model is trained using a dataset generated from the training sets of the evaluated DNN classifier. For a given video-type test, we generate four different types of features for the ranking model to make predictions: temporal features (TF), video embedding features (EF), prediction features (PF), and uncertainty features (UF). Ma *et al.* [20] previously demonstrated that test inputs located close to the decision boundary of the DNN classifier are more likely to be misclassified. Therefore, based on these four types of features, the ranking model can learn the test's proximity to the DNN classifier's decision boundary and, consequently, predict the probability of the test being misclassified by the model. We rank all test inputs in the target test set based on their misclassification probabilities. Videos with a higher likelihood of being misclassified are considered more likely to reveal faults. Consequently, these potentially misclassified videos will be prioritized higher. In the following, we provide detailed information about the four types of features generated for a specific test input.

- **Temporal features(TF)** TF captures the unique temporal coherence inherent in a given video-type test. The primary objective of generating TF is to convert a video test into a low-dimensional vector by taking into account the temporal continuity of frames.
- **Video embedding features (EF)** EF captures the intrinsic information of a given video test input itself. More specifically, EF captures the temporal dimension of video data and is obtained using existing frame sampling techniques [68] that are specifically designed for video data.
- **Prediction features (PF)** PF captures the model's classification information for a test input. PF features are derived from the output of a DNN classifier and represent the confidence of a prediction result, as previously utilized in several studies [48, 6].
- **Uncertainty features (UF)** UF captures the uncertainty associated with the model's classification. UF features are generated by calculating the uncertainty scores assigned to each test input using existing uncertainty metrics, such as DeepGini [6].

VRank demonstrates applicability in various domains. For example, when evaluating a video classification model designed to identify accident videos captured by

highway cameras, VRank can be utilized to detect potentially misclassified video test cases within the test dataset. These video tests have a higher likelihood of uncovering bugs in the model. Through early labeling and diagnosis of these video tests, VRank can accelerate the model debugging process, minimizing the need for time and manual labeling efforts.

Moreover, prioritizing video-type test inputs can provide several benefits for developers in the context of DNN testing: **1) Save labeling time and cost:** Prioritizing video data for testing can save the cost of traditional manual labeling. Developers can quickly identify tests that are most likely to be incorrectly predicted by the model and label them, reducing the overall labeling cost. Videos typically contain numerous frames and continuous dynamic information, requiring a significant investment of time and manual effort for labeling. Test prioritization can help reduce the cost of manual labeling; **2) Rapidly uncover bugs in video models:** Test prioritization on video-type tests can help developers quickly identify tests that are more likely to be misclassified by the video classification model. These tests can efficiently aid in identifying bugs in the model; **3) Identify weight parameters causing prediction errors:** These potentially misclassified tests can also assist developers in efficiently analyzing which weight parameters in the model are responsible for causing prediction errors; **4) Fine-tuning of video models:** Through prioritizing video-type tests for rapid bug identification and quick recognition of weight parameters associated with causing prediction errors, developers can better perform model fine-tuning.

We conducted an empirical study to evaluate the performance of VRank based on 120 subjects. Here, a subject refers to a pair of video dataset and DNN model. We compare VRank with four test prioritization approaches compatible with video datasets and one baseline method, random selection. Furthermore, we evaluated the effectiveness of VRank in scenarios where noise is present during testing. Our experimental results demonstrate that VRank achieved better effectiveness over all the compared test prioritization approaches, with an average improvement of 5.76%∼46.51% on natural datasets and 4.26%∼53.56% on noisy datasets. We publish our dataset, results, and tools to the community on Github[1].

Our work has the following major contributions:

❶ **Approach.** We propose VRank, the first test prioritization approach that is specifically designed for video datasets. Specifically, VRank utilizes video-oriented feature generation and learning-to-rank techniques to rank the test inputs and prioritize potentially-misclassified video inputs.

❷ **Study** We conduct an extensive study involving 120 subjects, including natural and noisy test sets, to evaluate the performance of VRank. We compare VRank against existing test prioritization approaches and random selection. Our experimental results demonstrate the effectiveness of VRank.

❸ **Feature contribution analysis** We conducted a comprehensive analysis to assess the individual contributions of various feature types to the effectiveness of VRank. Our findings demonstrate that all four types of generated features, namely temporal features (TF), uncertainty features (UF), prediction features (PF), and video embedding features (EF), contribute to enhancing the effectiveness of VRank.

The remaining sections of our paper are organized as follows. Section 3.2 provides

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

the background for our work. Section 3.3 presents the specific details of the VRank approach we propose. Section 3.4 exhibits the design of our study. Section 3.5 presents the relevant details of the experiments and the analysis of the experimental results. Section 3.6 discusses the limitations and threats to the validity of our study. Section 3.7 presents the related work of our study. Finally, we conclude our paper in Section 3.8.

3.2 Background

3.2.1 DNNs and DNN Testing

Classification deep neural networks (DNNs) [69] are foundational to many applications of deep learning [70]. These networks are characterized by their multilayer architecture consisting of an input layer, one or more hidden layers, and an output layer. Each layer of a DNN comprises a set of interconnected neurons [71] that interconnect via weighted edges. A neuron is a computational unit that applies an activation function to the inputs and the weights of the incoming edges. The resulting output is then propagated to the next layer via the edges. During training, the DNN automatically learns the optimal weights of the edges using a large set of labeled training data. Once trained, the DNN can accurately classify an input object, such as an image or a video, into its corresponding class or category.

Ensuring the quality and reliability of DNN models is of paramount importance, and DNN testing [46, 6, 48, 72, 73, 74, 75] has emerged as a widely used approach to achieve this goal. Analogous to traditional software systems [76, 77, 78, 79, 80], DNN testing involves inputs and oracles. In the context of DNN testing, test inputs refer to the input that the model is expected to classify, which can take diverse forms depending on the specific task of the DNN under test, including images, natural language, or speech. Test oracles in DNN testing rely on manual labeling, whereby each input is manually labeled with ground truth by human annotators. By comparing the labeled ground truth and the predicted output of the DNN model, it is possible to assess the accuracy of the model in predicting the correct output for the given input.

3.2.2 DNNs for Video Classification

In recent years, the volume of multimedia content available on the Internet has increased exponentially, leading to an explosion in the number of videos being shared every minute. This rapid growth of video content has created a pressing need to analyze and understand these videos for a variety of applications, including search, recommendation, and ranking. Over the past few decades, the computer vision community [21] has focused on developing algorithms to address different video analysis problems, notably video classification. While significant progress has been made in feature learning using deep learning approaches in the image domain [81], pre-trained convolutional network (ConvNet) models [82] have been developed for generating image features. These features represent the activations of the network's last few fully-connected layers. However, applying these image-based deep features directly to videos is typically not feasible.

To overcome this issue, Tran *et al.* [22] proposed the use of deep 3D ConvNets to learn spatio-temporal features for video classification, leveraging large-scale video datasets. Building upon their previous work, Tran *et al.* [23] conducted an empirical

ConvNet architecture search to improve spatiotemporal feature learning, which outperformed C3D on several datasets, with faster inference time, smaller model size, and more compact representation. In their subsequent work, Tran *et al.* [14] investigated several forms of spatiotemporal convolutions for video analysis and their effects on action recognition. Moreover, Feichtenhofer *et al.* [24] proposed the SlowFast network for video recognition, which comprises a Slow pathway for capturing spatial semantics and a Fast pathway for capturing motion at a fine temporal resolution.

3.2.3 Test Input Prioritization for DNNs

Test input prioritization aims to rank the test inputs based on their likelihood of being incorrectly predicted by a DNN model. The literature has proposed two main categories of approaches for test input prioritization: coverage-based and confidence-based. Coverage-based approaches (e.g., CTM [11]) extend traditional software system testing methods to DNN testing. The research work conducted by Feng *et al.* [6] compared their proposed confidence-based approach DeepGini with numerous coverage-based approaches, demonstrating that DeepGini outperforms existing coverage-based techniques in prioritizing tests regarding both effectiveness and efficiency. Weiss *et al.* [9] further conducted an extensive investigation of several notable uncertainty-based metrics like Vanilla SM, Prediction-Confidence Score (PCS), and Entropy. These metrics have been demonstrated to be effective in test prioritization. While the aforementioned confidence-based approaches can be adapted to prioritize video test inputs, they fail to account for the distinct characteristics inherent in video data during the test prioritization process. In contrast, our proposed VRank explicitly considers the unique features of videos by utilizing a carefully designed feature generation strategy. By taking into account these video-specific features, VRank achieves higher prioritization effectiveness compared to the aforementioned uncertainty-based methods. Currently, Wang *et al.* [10] proposed PRIMA, which is based on mutation analysis and learning-to-rank. However, PRIMA is not applicable to video-oriented test prioritization because PRIMA's mutation rules are not adapted to video data.

3.3 Approach

3.3.1 Overview

Figure 3.1 illustrates the comprehensive outline of the sequential stages involved in our proposed VRank test prioritization approach. In the subsequent sections, we provide a more detailed description of each step.

❶ **Feature vector generation** Given a video test set T and the model M to be evaluated, in this step, VRank aims to generate a feature vector for each test $t \in T$. To this end, for each test, VRank generates four different types of features for it and combines these features into a final feature vector. The specific methods for feature generation can be found in Section 3.3.2. Furthermore, Section 3.3.2 also describes how to combine the generated four different types of features into a final feature vector.

❷ **Ranking model training** After obtaining the final feature vector for each test input $t \in T$, in this step, we aim to leverage a ranking model to predict the probability of each test being predicted incorrectly by the model M based on

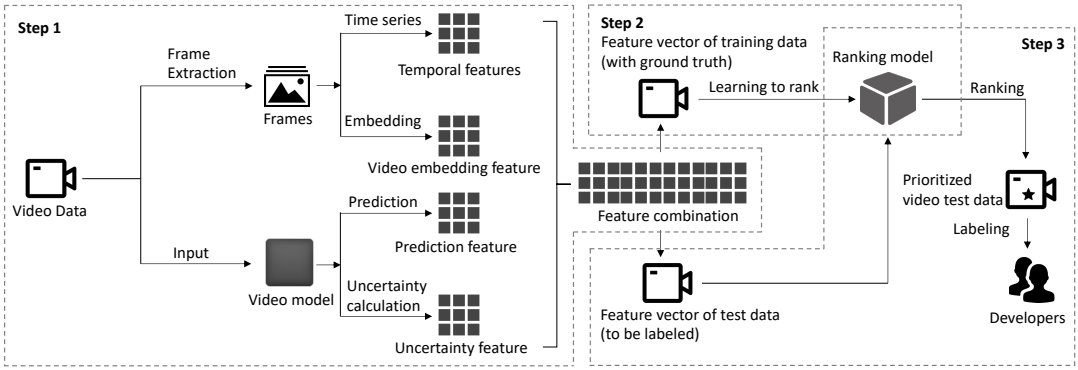


Figure 3.1: Overview of VRank

its final feature vector. The specific details regarding the training process of the ranking model and the methodology for utilizing the ranking model to predict misclassification probabilities can be found in Section 3.3.3.

❸ **Test prioritization** After obtaining the probability of each test (in the test set T) being misclassified by the model M using the ranking model, VRank utilizes this information for test prioritization. Tests with a higher probability of being misclassified will be prioritized higher. The specific details of this step can be found in Section 3.3.4.

In the subsequent sections, we provide a comprehensive description of each step outlined aforementioned, encompassing Video-oriented Feature Generation (cf. Section 3.3.2), Learning-to-rank (cf. Section 3.3.3), Variants of VRank (cf. Section 3.3.5), and the Usage of VRank (cf. Section 3.3.6). These sections delve into the intricate details of each step, offering a thorough understanding of the methodologies employed and their associated considerations.

3.3.2 Step 1: Video-oriented Feature Generation

Given a test set T of videos and a DNN model M to be tested, the objective of VRank is to prioritize tests that are more likely to be misclassified by the model M. VRank is based on video-oriented feature generation and the learning-to-rank technique. Therefore, in the first step, VRank generates four types of features for each video-type test input. In the following, we provide a comprehensive elucidation of the details for each type of feature, delving into the underlying rationale behind their inclusion in VRank and the methods employed for their generation. We aim to establish a clear understanding of their significance and relevance in the context of VRank.

- **Temporal features(TF)** TF captures the distinctive temporal coherence within a given video-type test. The primary aim of generating TF is to transform a video test into a low-dimensional vector by considering temporal frame continuity. In the following, we present the two main approaches we employed for generating TF features from consecutive frames: 1) Feature generation based on temporal changes. We compute variations between adjacent frames, encompassing Euclidean distance [83], Manhattan distance [84], squared difference distance [85], and Pearson similarity [86]. These metrics can indicate the extent of change between frames, effectively capturing the dynamic information of the video. 2) Statistical feature computation. We calculate statistical features such as variance, mean, and median for consecutive frames. These features contribute to delineating the individual characteristics of each frame.
- **Video embedding features (EF)** capture the intrinsic information of a given

video test input. To obtain EF, we employ existing frame sampling techniques [68] to extract a fixed number of frames from a given video-type test input t. We then utilize the pre-trained ResNet model [87] to map each frame into a vector representation. Finally, we compute the average of all frame vectors to obtain a representative vector for the entire video.

- **Prediction features (PF)** captures the model's classification information for a test input. To obtain PF, we input t into the model M, and M will output a probability vector representing the probabilities of t belonging to each class. For example, a feature vector $\{0.2, 0.3, 0.5\}$ signifies that, according to the predictions made by model M, the test input t has a 20% probability of belonging to class 1, a 30% probability of belonging to class 2, and a 50% probability of belonging to class 3. PF has been utilized in various prior studies [48, 6].
- **Uncertainty features (UF)** captures the uncertainty associated with the model's classification. To obtain UF, we leverage six existing uncertainty metrics [9, 6, 88] (i.e., DeepGini, Vanilla SM, PCS, Entropy, Margin, and Least Confidence) to obtain a set of uncertainty scores for each test input t. These metrics have been widely recognized for their outstanding effectiveness in quantifying uncertainty in classification tasks. For each test input t, we compute the corresponding uncertainty scores using each of the six metrics. These scores represent the model's uncertainty in predicting the class for t. The UF vector for a given test input is then constructed by concatenating the six uncertainty scores, resulting in a vector representation: $\{S_1, S_2, S_3, S_4, S_5, S_6\}$. Each element S_i represents the uncertainty associated with the model's prediction for the test input t calculated by the i_{th} uncertainty-based metric.

For each test input $t \in T$, VRank combines its four aforementioned types of features to generate a comprehensive and representative feature vector. This feature vector encapsulates the relevant information from all feature types for the given test input.

Below, we explain how the aforementioned features contribute to determining the decision boundaries of the model:

- **Temporal features (TF)** Temporal features can capture the unique temporal coherence in a given video type test. Generating time features allows the transformation of video tests into low-dimensional vectors, where the model's decision boundaries can be perceived as a geometric interface. Low-dimensional video vectors, when mapped into space, can indirectly reflect the distance between the video-type test and the decision boundary.
- **Video embedding features (EF)** These features can effectively capture the intrinsic information of the video test input, particularly the temporal dimension of video data. Through this capture, the video input can be mapped to a spatial vector, where the model's decision boundaries can be seen as a geometric interface. The numerical video embedding feature can facilitate the calculation of the distance between a video-type test and the decision boundary. Therefore, the embedding feature can indirectly reflect the proximity between a test and the decision boundaries.
- **Prediction features (PF)** These features originate from the DNN classifier's classification information for the test input. PF features reflect the model's confidence in the prediction results and can be used to evaluate the model's accuracy in predicting specific test inputs. If a test input's PF features indicate

that the model is not confident in its classification result, it can suggest that the input is close to the model's decision boundaries.

- **Uncertainty features (UF)** These features represent the model's uncertainty about its classification decisions. By calculating uncertainty scores for each test input (e.g., using DeepGini), UF features can assist in identifying test inputs for which the model exhibits higher uncertainty during classification. Test inputs with high uncertainty are more likely to be located near the model's decision boundaries.

Below, through a specific example, we illustrate how VRank integrates the aforementioned four types of features into a final feature vector. Assuming that, for a given video-type test input, VRank generates four types of features for it: temporal features (TF) of i dimensions, denoted as (v_1, v_2, ..., v_i), embedding features (EF) of j dimensions, denoted as (e_1, e_2, ..., e_j), prediction features (PF) of k dimensions, denoted as (p_1, p_2, ..., p_k), and uncertainty features (UF) of n dimensions, denoted as (u_1, u_2, ..., u_n). VRank combines these four types of vectors by concatenation, producing a final vector of $(i + j + k + n)$ dimensions: (v_1, v_2, ...v_i, e_1, e_2, ..., e_j, p_1, p_2, ..., p_k, u_1, u_2, ...,u_n).

In the following section, we provide a detailed explanation of the methodology employed to obtain the misclassification score.

3.3.3 Step 2: Learning-to-rank

In this step, we employ the ranking model LightGBM [89] to learn from the feature vector of $v \in V$ to predict its misclassification score. LightGBM is an advanced gradient-boosting framework renowned for its ability to learn features for efficient classifications. We follow the process below to train the LightGBM model: Given the video classification M with the dataset used for its evaluation, we initially partition the dataset into two sets: the training set R and the test set T. The test set is kept untouched for evaluating VRank. Our objective is to construct a training set R' for training the ranking models based on the training set R. To achieve this, we generate the final feature vector for each $r \in R$ by following the steps in Section 3.3.2. These features serve as the training features for the dataset R. Subsequently, we employ the original video classification model M to classify each instance $r \in R$, aiming to identify whether each r is misclassified by the model M. If r is misclassified, it will be labelled as 1; otherwise, it will be labelled as 0. Consequently, we obtain the labels for the training set R. Using the constructed training set, we train the LightGBM ranking model for VRank.

3.3.4 Step 3: Test Prioritization

The LightGBM ranking model, trained in the previous step, was originally designed for binary classification, classifying a given input into one of two classes. However, our objective is to obtain a misclassification probability score for each test input, indicating the likelihood of it being misclassified by the model. To achieve this, we applied specific adjustments to the original LightGBM model: We extract the intermediate value from the model's output for a given input, which can indicate the misclassification probability. Typically, in the model prediction process, if this intermediate value exceeds a predefined threshold, the input is labeled as "misclassified"; otherwise, it is labeled as "not misclassified". Instead of proceeding with the final classification, we directly employ this intermediate value as

the misclassification probability score. A higher score implies a greater probability of the test instance being misclassified. Finally, we rank all tests in the test set T in descending order based on their misclassification probability scores.

3.3.5 Variants of VRank

We investigate the impact of different ranking models on the effectiveness of VRank and propose three variants of VRank, namely VRank^X, VRank^R, and VRank^L. These variants employ the XGBoost [13], Random Forest [90], and Logistic Regression [91] respectively, as their underlying ranking models. It is important to note that the execution workflow of these variants closely resembles that of VRank, and the sole distinction lies in the selection of ranking models.

Additionally, we also extended the adjustments made to the ranking model LightGBM of VRank to the ranking models of VRank's variants. Specifically, for a test input, rather than having the ranking models output a binary classification (i.e., whether the test will be predicted incorrectly by the model), we extract the intermediate output to obtain the probability of this test being misclassified. In this way, we can obtain the misclassification score of each test input, which can be utilized for test prioritization. In the following, we provide a detailed explanation of the specific ranking models utilized by each variant of VRank.

- **VRank^X** In the context of VRank^X, we leverage the XGBoost ranking algorithm [13] to predict the misclassification score associated with a given test input, based on its feature vector. XGBoost is a powerful gradient-boosting technique that effectively integrates decision trees to augment prediction accuracy.
- **VRank^R** In the context VRank^R, we adopt Random Forest [90] as the ranking model. Random forest is an ensemble learning algorithm that constructs multiple decision trees. The predictions from individual trees are combined to produce the final prediction using averaging or voting.
- **VRank^L** In the context VRank^L, we adopt Logistic Regression [91] as the ranking algorithm. Logistic Regression is a statistical modeling technique that uses a logistic function to model the association between a categorical dependent variable and one or more independent variables.

3.3.6 Usage of VRank

Utilizing ranking models, VRank is capable of predicting a misclassification score for each test input within a designated test set. Test inputs with higher scores are assigned a higher priority. The ranking models employed in VRank undergo pre-training prior to their execution, following standardized and consistent procedures. In the subsequent parts, we comprehensively present the training process, outlining the specific steps taken to train the ranking models.

Given a video dataset and the model M under test, the initial step is to partition the dataset into two subsets: the training set R and the test set T, with a ratio of 7:3 [92]. The test set T remains untouched to evaluate the effectiveness of VRank. Based on the training set R, our objective is to construct a new training set R' specifically for training the ranking models. Initially, a feature vector F_v is generated for each input $r \in R$. The generation procedures for the feature vector are described in Section 3.3.2. These feature vectors are then used to construct a new training set R'. To obtain the labels for each sample in R', we input $r_i \in R$ into the model M. Leveraging the known ground truth of the training set R if r_i is incorrectly predicted

by model M, the label of the corresponding $r'_i \in R'$ is set to 1; otherwise, it is set to 0.

Based on the constructed training set R', we train the ranking models. Upon the completion of the training process, the ranking models are capable of predicting the likelihood of misclassification for a given test input based on its corresponding feature vector.

3.4 Study design

3.4.1 Research questions

Our experimental evaluation answers the research questions below.

- **RQ1: How does VRank perform in prioritizing video test inputs?**
  We assess the effectiveness and efficiency of VRank and compare it with multiple existing testing prioritization approaches, including DeepGini, Vanilla Softmax, PCS, Entropy, and random selection.
- **RQ2: How does VRank perform on noisy video data?**
  To evaluate the effectiveness of VRank in noisy contexts, we employ a range of noise generation techniques derived from prior research works [93, 94, 95, 96] to generate video datasets with simulated noise. We compare VRank's effectiveness on these generated noisy datasets with the aforementioned test prioritization approaches to demonstrate its effectiveness.
- **RQ3: What is the impact of different ranking models on the effectiveness of VRank?**
  Within the learning-to-rank process of VRank, we employed the LightGBM [89] ranking algorithm. In this research question, we introduce three variants of VRank by modifying the ranking models to Random Forest [90], XGBoost [13], and Logistic Regression [91], respectively. By evaluating the effectiveness of VRank and its variants, we aim to explore which ranking algorithm can better utilize the generated features for test prioritization.
- **RQ4: To what extent do each type of features contribute to the effectiveness of VRank?**
  In VRank, we generate four distinct types of features from each test input for test prioritization, namely temporal features (TF), video embedding features (EF), prediction features (PF), and uncertainty features (UF), as elaborated in Section 6.3. In this research question, we focus on comparing the contributions of the three types of features toward the effectiveness of VRank.
- **RQ5: What is the influence of the number of extracted frames on the effectiveness of VRank?**
  Two critical steps in VRank are to generate video embedding features and temporal features from a given test to predict the likelihood of the test being misclassified. To obtain these two types of features, we utilize established frame sampling techniques [68] to extract a fixed number of frames from the video-type test input. In this research question, we explore the impact of the number of extracted frames on the effectiveness of VRank.

3.4.2 Subjects

The effectiveness of VRank and the compared test prioritization approaches [6, 9] was evaluated using a set of 120 subjects, where each subject corresponds to a video

Table 3.1: Video models and datasets

ID	Dataset	# Videos	Model	Type
1	UCF101	13320	C3D	Original, HF, HS, WS, FSN, SR, ZCA, CSR
2	UCF101	13320	R2Plus1D	Original, HF, HS, WS, FSN, SR, ZCA, CSR
3	UCF101	13320	R3D	Original, HF, HS, WS, FSN, SR, ZCA, CSR
4	UCF101	13320	SlowFastNet	Original, HF, HS, WS, FSN, SR, ZCA, CSR
5	UCF101	13320	VT	Original, HF, HS, WS, FSN, SR, ZCA, CSR
6	HMDB51	6849	C3D	Original, HF, HS, WS, FSN, SR, ZCA, CSR
7	HMDB51	6849	R2Plus1D	Original, HF, HS, WS, FSN, SR, ZCA, CSR
8	HMDB51	6849	R3D	Original, HF, HS, WS, FSN, SR, ZCA, CSR
9	HMDB51	6849	SlowFastNet	Original, HF, HS, WS, FSN, SR, ZCA, CSR
10	HMDB51	6849	VT	Original, HF, HS, WS, FSN, SR, ZCA, CSR
11	HWID12	2782	C3D	Original, HF, HS, WS, FSN, SR, ZCA, CSR
12	HWID12	2782	R2Plus1D	Original, HF, HS, WS, FSN, SR, ZCA, CSR
13	HWID12	2782	R3D	Original, HF, HS, WS, FSN, SR, ZCA, CSR
14	HWID12	2782	SlowFastNet	Original, HF, HS, WS, FSN, SR, ZCA, CSR
15	HWID12	2782	VT	Original, HF, HS, WS, FSN, SR, ZCA, CSR

dataset with a model. Essential details regarding these subjects are presented in Table 3.1. Specifically, the "#Videos" column indicates the number of videos in a dataset, while the "Type" column denotes the dataset's type. "Original" denotes natural data, while other non-original types are abbreviations representing different types of noise. For instance, "HF" indicates Horizontal Flip noise.

Among the 120 subjects, 15 subjects (3 video datasets × 5 models) were generated using natural datasets, while the remaining 105 subjects were generated using noisy datasets. To generate the noisy datasets, we applied 7 noise generation techniques to each natural dataset, resulting in 7 noisy datasets. Each noisy dataset was then paired with 5 models. Therefore, the total number of subjects is 105 (3 video datasets × 5 models × 7 noise generation techniques). In the subsequent section, we present a comprehensive description of the datasets and models employed in our research.

3.4.2.1 DNN Models

We assess the effectiveness of VRank based on five prevalent video classification models: C3D [22], R3D [23], R2Plus1D [14], SlowFast [24] and VT [97]. The reason we selected these models for evaluating VRank is that: 1) These models are widely recognized in the field of video classification and have extensive applications in both academia and industry [22, 23, 14, 24]. 2) Each model has its unique architecture and approach to handling video data. 3) Since these models have undergone extensive testing and application on multiple datasets [22, 23, 14, 24], they provide VRank with a solid benchmark for effectively comparing VRank's performance across different models.

Although we only conduct tests on these specific models, it is important to note that VRank can be applied to a wide range of video classification models.

- **C3D** [22] The C3D (Convolutional 3D) network is an architecture of 3D Convolutional Networks designed to learn spatio-temporal data, particularly in the form of videos. C3D comprises eight convolutional layers, five max-pooling layers, and two fully connected layers, followed by a softmax output layer. C3D's unique ability to model both appearance and motion information simultaneously is a key factor in its superior performance compared to 2D ConvNet features on various video analysis tasks. This is because videos are inherently spatio-temporal and therefore require specialized architectures capable of extracting and processing

information in all three dimensions.

- **R3D** [23] The R3D network is a variant of 3D Convolutional Networks, incorporating design elements from both ResNet [87] and C3D architectures. Specifically, R3D leverages residual connections from ResNet, which facilitate the training of DNNs by allowing gradients to flow directly through the network. Additionally, the R3D architecture uses 3D ConvNets to learn spatiotemporal features, making it particularly suited for video-based tasks.
- **R2Plus1D** [14] The R2Plus1D architecture effectively addresses the computational complexity associated with action recognition tasks by decomposing the 3D convolutions into a fusion of spatial and temporal convolutions. This decomposition enables more efficient utilization of computational resources compared to fully 3D convolutions.
- **SlowFast** [24] SlowFast is a video recognition architecture that introduces two pathways, namely the slow pathway and the fast pathway. The slow pathway effectively functions at a reduced frame rate, thereby facilitating the extraction and analysis of spatial semantics pertaining to the video content. Conversely, the fast pathway operates at a significantly higher frame rate, affording the capacity to capture motion nuances with exceptional temporal resolution.
- **VT** [97] The Video Classification with Transformers (VT) model is an open-source project officially released by the Keras deep learning framework. It integrates Convolutional Neural Networks (CNNs) and Transformers to enhance video classification capabilities. Specifically, it employs CNNs to extract frame-level features from the video and then feeds these features into a Transformer to capture temporal relationships between different frames. The model is designed to effectively learn spatial-temporal features from video data, enhancing its ability to classify video content accurately. This approach showcases an advanced application of deep learning in video analysis.

3.4.2.2 Datasets

In our study, we assess the performance of VRank using three widely-adopted video datasets: HWID12 [98], HMDB51 [99], and UCF101 [100]. The rationale behind choosing these three datasets is their extensive usage in the realm of video classification. More specifically, we select these datasets for evaluating VRank due to the following two reasons: **1) Diversity and Representativeness** These three datasets exhibit diversity and representativeness in the context of video classification. HWID12 includes real-world surveillance videos of high-speed highway traffic, HMDB51 covers various daily actions, and UCF101 contains a variety of action videos from the real world. This diversity ensures that VRank be evaluated more comprehensively; **2) Wide Applications and Recognition** These datasets are widely used and recognized in the fields of computer vision and video analysis [98, 99, 100]. Utilizing these well-established datasets can enhance the generalizability of VRank's evaluation results.

- **HWID12** [98] The HWID12 dataset serves for the classification task of real-time highway accident detection in intelligent transportation systems. HWID12 comprises 2,782 video clips with duration ranging from 3 to 8 seconds, categorized into twelve classes (e.g., "Sideswipe collision", "Collision with motorcycle" and "Pedestrian hit").

- **HMDB51** [99] The HMDB51 dataset comprises video clips sourced from movies, supplemented by a small portion obtained from public databases such as the Prelinger Archives, YouTube, and Google Videos. HMDB51 is composed of 6,849 videos, classified into 51 action categories (e.g., "Drink", "Hug", and "Walk"), with each category containing at least 100 clips.
- **UCF101** [100] The UCF101 dataset is an action recognition dataset collected from YouTube. UCF101 consists of 13,320 videos, categorized into 101 action classes (e.g., "High Jump", "Punch", and "Diving").

3.4.3 Noise generation techniques

In our study, we employed seven noise generation techniques to generate video inputs with noise. These techniques were selected based on prior research studies [93, 94, 95, 96]. The following is a description of each technique:

- **Channel Shift (CSR):** CSR applies modifications to the overall color representation of a video by shifting the value of the color channel. This technique introduces color perturbations by adding random noise to each pixel's color channel values, thus altering the color appearance of the video.
- **Feature-wise Normalization (FSN):** FSN performs normalization of the features in each video input by dividing it with the standard deviation. This process aims to decentralize the video dataset and normalize the feature distributions, enabling the model to capture finer-grained variations in the data.
- **Height Shift (HS):** HS vertically displaces a given video by a certain number of pixels, effectively shifting its position up or down within the frame. This augmentation technique introduces spatial transformations, such as simulating camera movements or object repositioning, by adding random noise to the vertical position of each frame.
- **Width Shift (WS):** WS horizontally shifts the position of a video input by a specified number of pixels. By applying random horizontal offsets to each frame, WS enables the model to learn robustness to variations in object positioning and enhances its ability to handle objects appearing at different spatial locations within the frame.
- **Shear (SR):** SR refers to the intentional distortion of a video along its axes with the primary objective of creating or correcting perceptual angles.
- **Horizontal Flip (HF):** HF horizontally flips a given video by mirroring the content along the vertical axis. This operation introduces left-right orientation changes to the video frames, augmenting the dataset with horizontally flipped versions of the original videos.
- **ZCA Whitening (ZCA):** ZCA whitening applies dimension reduction operations to the given videos, reducing redundant information while preserving crucial features. By performing a linear transformation on the pixel values of each frame, ZCA whitening removes correlations between neighboring pixels, effectively decorrelating the data and enhancing the model's ability to focus on meaningful variations in the video content.

3.4.4 Compared Approaches

To demonstrate the effectiveness of VRank, we compare it with five distinct test prioritization approaches, including a baseline approach, namely random selection, alongside four DNN test prioritization techniques. The rationale behind selecting

these particular methods rests on three key factors: Firstly, their adaptability to facilitate test prioritization on video datasets, which is a pivotal requirement for our research context. Secondly, their effectiveness in the context of DNNs has been well demonstrated in the existing literature [6, 9, 101]. Lastly, the availability of open-source implementations. All of the selected approaches are accessible for implementation purposes.

- **DeepGini** [6] employs the Gini coefficient to measure the likelihood of misclassification, thereby enabling the ranking of test inputs. The calculation of Gini score is presented in Formula 4.8.

$$\xi(x) = 1 - \sum_{i=1}^{N} \left(p_i(x)\right)^2 \tag{3.1}$$

  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.
- **Vanilla SM** [9] calculates the difference between the value of 1 and the maximum activation probability in the output softmax layer. Formula 6.21 provides a clear depiction of the calculation process.

$$\mathrm{V}(x) = 1 - \max_{c=1}^{C} l_c(x) \tag{3.2}$$

  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 [9] quantifies the level of uncertainty in a classification model's prediction for a given test by computing the difference between the predicted class and the second most confident class. PCS is calculated by Formula 4.10.

$$P(x) = l_k(x) - l_j(x) \tag{3.3}$$

  where $l_k(x)$ refers to the most confident prediction probability. $l_j(x)$ refers to the second most confident prediction probability.
- **Entropy** Entropy [9] measures uncertainty in a classification model's prediction for a given test by computing the entropy of the softmax likelihood.
- **Random selection** [102] In random selection, the order of execution for test inputs is determined randomly.

3.4.5 Measurements

Following the prior research on DNN test prioritization [6], we employ the Average Percentage of Fault-Detection (APFD) [11] metric to assess the effectiveness of VRank and the compared approaches. APFD is a well-established and widely accepted measure for evaluating prioritization strategies. Generally, higher APFD scores indicate a faster rate of misclassification detection. We determine the APFD values by utilizing Formula 6.17.

$$APFD = 1 - \frac{\sum_{i=1}^{k} o_i}{kn} + \frac{1}{2n} \tag{3.4}$$

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 DNN model M. 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. Below, we explain the rationale for using APFD to assess the effectiveness of a test prioritization method in detecting misclassified tests: In the formula of APFD (Formula 6.17), a smaller $\sum_{i=1}^{k} o_i$ suggests that the misclassified tests are positioned relatively closer to the front of the prioritized test set. This implies that the prioritization approach effectively places misclassified tests at the beginning of the test set, indicating a higher level of effectiveness. Consistent with previous research [6], we normalize the APFD values to the range [0,1]. A prioritization approach is deemed more effective when the APFD value is closer to 1.
Efficiency measurement of VRank: Following the existing study [10], we evaluate the efficiency of VRank by quantifying the time required for each step of VRank, as well as the time cost of each compared approach.

3.4.6 Implementation and Configuration

VRank was implemented in Python utilizing PyTorch 2.0.0 [103], OpenCV 4.7.0, and scikit-learn 1.0.2 libraries. In terms of the compared approaches [6, 9], we integrated existing implementations of them into our experimental pipeline. In terms of ranking models, for XGBoost and LightGBM, we employed the specific versions XGBoost 1.7.4 and LightGBM 3.3.5. For the ranking model random forest and logistic regression, we leveraged the existing algorithm packages provided by scikit-learn. Concerning the parameter configurations, we set the *n_estimators* parameter to 100 for the XGBoost, LightGBM, and Random Forest ranking algorithms. For the Logistic Regression ranking algorithm, we set the *max_iter* parameter to 100. Our experiments were conducted on NVIDIA Tesla V100 32GB GPUs. In terms of data analysis, the corresponding experiments were performed on a MacBook Pro laptop with Mac OS Big Sur 11.6, Intel Core i9 CPU, and 64 GB RAM.

3.5 Results and analysis

3.5.1 RQ1: Effectiveness and efficiency of VRank

Objective: We investigate the effectiveness and efficiency of VRank, comparing it with existing test input prioritization approaches and random selection.
Experimental design: We employed 12 pairs of video datasets and models as subjects in our study to assess the effectiveness of VRank. The fundamental details of these datasets and models can be found in Table 3.1. Specifically, we selected five compared approaches, consisting of four test prioritization techniques (namely DeepGini, Vanilla SM, PCS, and Entropy) along with a baseline approach (random selection). We utilize these approaches for comparison because they can be adapted to prioritize testing on video datasets. To quantify the effectiveness of each approach, we employed the Average Percentage of Fault-Detection (APFD), a widely accepted measure in the field. In addition to assessing effectiveness, we investigated the efficiency of VRank by analyzing the time required for each step of its execution and comparing its overall execution time with that of the five compared approaches.

Furthermore, due to the randomness of the model training process, we performed a statistical analysis to ensure the stability of our findings. Specifically, we repeated all experiments ten times for each subject and reported the average results. Furthermore, we calculated the p-value of the experiments to assess whether the VRank approach

Table 3.2: Effectiveness comparison among VRank, Random, DeepGini, VanillaSM, PCS, and Entropy in terms of the APFD values on natural datasets

Data	Model	Approach					
		Random	DeepGini	VanillaSM	PCS	Entropy	**VRank**
HWID12	C3D	0.513	0.717	0.716	0.712	0.717	0.745
	R2Plus1D	0.492	0.695	0.696	0.685	0.694	0.735
	R3D	0.488	0.696	0.698	0.697	0.694	0.742
	SlowFastNet	0.528	0.703	0.701	0.698	0.705	0.744
	VT	0.522	0.721	0.723	0.726	0.718	0.746
HMDB51	C3D	0.484	0.661	0.659	0.653	0.663	0.702
	R2Plus1D	0.495	0.573	0.577	0.577	0.568	0.616
	R3D	0.493	0.619	0.623	0.622	0.609	0.658
	SlowFastNet	0.488	0.614	0.614	0.612	0.614	0.650
	VT	0.503	0.704	0.708	0.706	0.696	0.735
UCF101	C3D	0.501	0.759	0.758	0.754	0.758	0.817
	R2Plus1D	0.503	0.766	0.766	0.764	0.763	0.806
	R3D	0.498	0.697	0.697	0.693	0.694	0.755
	SlowFastNet	0.492	0.717	0.716	0.711	0.720	0.772
	VT	0.507	0.749	0.751	0.749	0.743	0.797

Table 3.3: Performance improvement of VRank on the 15 initial subjects (i.e., three natural input sets on 5 Video classification models)

Approach	**# Best cases**	**Average APFD**	**Improvement(%)**
Random	0	0.501	46.51
DeepGini	0	0.692	6.07
VanillaSM	0	0.694	5.76
PCS	0	0.691	6.22
Entropy	0	0.690	6.38
VRank	15	0.734	-

consistently outperformed the compared approaches.

To further illustrate the statistical significance of the improvements in VRank compared to other test prioritization approaches, we conducted a statistical analysis by calculating p-values and effect size associated with the experimental results. Regarding the calculation of p-values, we employed the **paired two-sample t-test** [104], which is a widely used statistical method for evaluating differences between two related datasets. If the p-value is less than 10^{-05}, it indicates that the difference between the two sets of data is statistically significant [105]. For the measurement of effect size, we utilized Cohen's d for measuring the effect size [106]. In this context, values of $|d| < 0.2$ are categorized as "negligible," $|d| < 0.5$ as "small," $|d| < 0.8$ as "medium," and otherwise as "large". For instance, if we compare VRank with another test prioritization method, and the value of d is 0.7, the effect size is categorized as "medium" because $0.5 < 0.7 < 0.8$. This suggests that there is a relatively medium difference between the two methods.

Table 3.4: Statistical analysis on natural test inputs (in terms of p-value and effect size)

	Random	DeepGini	VanillaSM	PCS	Entropy
VRank (p-value)	1.755×10^{-09}	7.336×10^{-07}	1.023×10^{-06}	1.605×10^{-06}	6.311×10^{-07}
VRank (effect size)	7.615	3.816	3.669	3.479	3.884

Results: The experimental findings pertaining to RQ1 are presented in Table 3.2, Table 3.3, Table 3.4 and Table 3.5. We highlight the approach with the highest effectiveness in grey to facilitate quick and easy interpretation of the results. Table 3.2 presents the effectiveness of VRank and the compared approaches across different

Table 3.5: Time cost of VRank and the compared approaches

Time cost	Approach					
	VRank	Random	DeepGini	VanillaSM	PCS	Entropy
Feature generation	2.3 min	-	-	-	-	-
Ranking model training	35 s	-	-	-	-	-
Prediction	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s

video subjects, as measured by the Average Percentage of Faults Detected (APFD). We see that VRank performs better than all the compared approaches regarding APFD across all cases. Specifically, the APFD range for VRank spans from 0.616 to 0.817, whereas the baseline approach exhibits an APFD range of 0.484 to 0.528. Furthermore, the uncertainty-based test prioritization methods yield an APFD range of 0.563 to 0.766. Table 3.3 provides a detailed analysis of the experimental results for RQ1, focusing on three aspects: the number of best-performing cases for each test prioritization approach, the average effectiveness, and the relative improvement of VRank compared to each method. It is observed that the average APFD of VRank is 0.734, with an average improvement of 5.76%∼46.51% compared to the uncertainty-based test prioritization approaches and random selection. Based on the aforementioned results, we conclude that VRank exhibits better effectiveness in prioritizing video test inputs compared to DeepGini, VanillaSM, PCS, Entropy, and Random Selection.

Table 3.4 presents the results of the statistical analysis evaluating the improvement of VRank in comparison to other test prioritization methods. The analysis employs two key metrics: p-value and effect size. As mentioned in the experimental design above, a p-value below 10^{-05} indicates that the difference between two datasets [105] is statistically significant. An effect size of ≥ 0.8 suggests that the difference in effectiveness between the two approaches is considered "large".

In Table 3.4, we see that all the p-values between VRank and other test prioritization approaches consistently fall below 10^{-05}. This suggests that VRank significantly outperforms all the test prioritization methods being compared. For example, the p-value between VRank and DeepGini is 7.336×10^{-07}, while the p-value between VRank and VanillaSM is 1.023×10^{-06}. Moreover, the effect sizes between VRank and all the compared approaches exceed 0.8, suggesting that the improvement in VRank's effectiveness (measured by APFD) compared to all the other approaches is "large". For instance, the effect size between VRank and VanillaSM is 3.669, and the effect size between VRank and DeepGini is 3.816.

Table 3.5 provides a comprehensive breakdown of the time required by each step of VRank and a comparison with uncertainty-based test prioritization approaches and random selection. The time required for VRank is partitioned into three steps: feature generation, ranking model training, and prediction. Our findings reveal that feature generation is the most time-consuming step, taking approximately 2.3 minutes, followed by ranking model training, which takes approximately 35 seconds. Notably, the prediction time of VRank is fast, taking less than 1 second once the ranking model is trained and the features have been generated. Overall, the average time consumption of VRank for each dataset is approximately 3 minutes. Although VRank is less efficient than uncertainty-based test prioritization approaches, which take less than 1 second, its time cost is acceptable compared to the prohibitively expensive manual labeling.

Answer to RQ1: When applied to natural datasets, VRank demonstrates better effectiveness in prioritizing video test inputs compared to DeepGini, VanillaSM, PCS, Entropy, and Random Selection, with the average improvement of 5.76%∼46.51%. Furthermore, the total time required for the execution of VRank is less than 3 minutes, which falls within an acceptable range.

3.5.2 RQ2: Effectiveness on noisy test inputs

Objective: We evaluated the effectiveness of VRank on noisy test inputs. To this end, we incorporated various types of video noise, namely Channel Shift (CSR), Feature-wise Normalization (FSN), Height Shift (HS), Width Shift (WS), Shear (SR), Horizontal Flip (HF), and ZCA Whitening (ZCA), as discussed in Section 3.4.3. We derived inspiration for these noise types from prior research [93, 94, 95, 96].

Experimental design: In order to generate noisy video datasets, we employed seven noise generation techniques, namely Channel Shift (CSR), Feature-wise Normalization (FSN), Height Shift (HS), Width Shift (WS), Shear (SR), Horizontal Flip (HF), and ZCA Whitening (ZCA). By applying these techniques, we introduced various forms of noise and perturbations to the original video datasets, thereby increasing their diversity and complexity. In total, we constructed 84 subjects for evaluation (4 video models $\times$ 3 video datasets $\times$ 7 noise generation techniques). Consistent with our previous research question, we compared VRank with four test prioritization approaches and a baseline method (i.e., random selection), using the metric APFD to quantify their effectiveness.

Results: The experimental results for RQ2 are presented in Table 3.6 Table 3.7 and Table 3.8. Table 3.6 showcases the effectiveness of VRank in comparison to several test prioritization techniques and the baseline (i.e., random selection) across noisy datasets generated using various noise generation techniques. We see that VRank consistently performs better than all the compared methods in terms of average APFD across all cases. More specifically, the average APFD of VRank ranges from 0.612 to 0.758, while the baseline method exhibits an average APFD ranging from 0.490 to 0.518. The uncertainty-based test prioritization techniques achieve an average APFD between 0.555 to 0.728. Overall, VRank demonstrates an improvement ranging from 4.26% to 53.56% compared with DeepGini, VanillaSM, PCS, Entropy, and Random Selection. This improvement is consistently observed across each noise generation technique. For instance, under the HS noise technique, VRank exhibits an improvement ranging from 5.32% to 43.17%. Similarly, under the HF noise technique, the improvement ranges from 4.80% to 46.26%, and under the CSR noise technique, it ranges from 4.26% to 48.28%.

Table 3.7 provides a detailed analysis of the experimental results for RQ2, focusing on three aspects: the number of best-performing cases for each test prioritization approach, the average effectiveness, and the relative improvement of VRank compared to each method. We see the average APFD of VRank is 0.692, with an average improvement of 7.12% to 38.68% compared to other test prioritization approaches.

In Table 3.8, we present a detailed analysis of VRank's effectiveness by using the SR noise technique as an example. We can see that VRank consistently performs better than the compared approaches across all subjects (a DNN model associated with a noisy dataset) related to SR. Moreover, the APFD values of VRank range from 0.577 to 0.756, while that of the compared approaches range from 0.496 to 0.712. The aforementioned experimental results indicate that VRank maintains

Table 3.6: Overall effectiveness comparison on noisy video datasets

Noise Data	Approach	Average APFD					Improvement(%)				
		C3D	R2Plus1D	R3D	SlowFastNet	VT	C3D	R2Plus1D	R3D	SlowFastNet	VT
HF	Random	0.502	0.507	0.505	0.504	0.508	37.85%	31.36%	32.67%	30.75%	46.26%
	DeepGini	0.651	0.624	0.618	0.606	0.707	6.30%	6.73%	8.41%	8.75%	5.09%
	VanillaSM	0.651	0.625	0.622	0.606	0.709	6.30%	6.56%	7.72%	8.75%	4.80%
	PCS	0.647	0.616	0.620	0.605	0.708	6.96%	8.12%	8.06%	8.93%	4.94%
	Entropy	0.651	0.623	0.612	0.606	0.703	6.30%	6.90%	9.48%	8.75%	5.69%
	VRank	0.692	0.666	0.670	0.659	0.743	-	-	-	-	-
HS	Random	0.505	0.496	0.498	0.503	0.498	24.16%	30.44%	33.73%	21.67%	43.17%
	DeepGini	0.586	0.602	0.611	0.557	0.676	7.00%	7.48%	9.00%	9.87%	5.47%
	VanillaSM	0.585	0.604	0.616	0.557	0.677	7.18%	7.12%	8.12%	9.87%	5.32%
	PCS	0.580	0.597	0.615	0.555	0.673	8.10%	8.38%	8.29%	10.27%	5.94%
	Entropy	0.586	0.601	0.604	0.558	0.672	7.00%	7.83%	10.26%	9.68%	6.10%
	VRank	0.627	0.647	0.666	0.612	0.713	-	-	-	-	-
WS	Random	0.498	0.506	0.506	0.499	0.499	33.53%	30.83%	31.42%	27.45%	46.49%
	DeepGini	0.627	0.620	0.614	0.591	0.694	6.06%	6.77%	8.31%	7.61%	5.33%
	VanillaSM	0.627	0.621	0.617	0.590	0.694	6.06%	6.60%	7.78%	7.80%	5.33%
	PCS	0.621	0.611	0.617	0.587	0.690	7.09%	8.35%	7.78%	8.35%	5.94%
	Entropy	0.627	0.618	0.608	0.592	0.693	6.06%	7.12%	9.38%	7.43%	5.48%
	VRank	0.665	0.662	0.665	0.636	0.731	-	-	-	-	-
FSN	Random	0.491	0.494	0.501	0.508	0.507	53.56%	45.55%	43.60%	42.13%	49.70%
	DeepGini	0.712	0.678	0.670	0.678	0.725	5.90%	6.05%	7.16%	6.49%	4.69%
	VanillaSM	0.711	0.680	0.672	0.677	0.728	6.05%	5.74%	6.85%	6.65%	4.26%
	PCS	0.706	0.676	0.671	0.674	0.728	6.80%	6.36%	7.00%	7.12%	4.26%
	Entropy	0.713	0.675	0.666	0.679	0.719	5.75%	6.52%	7.81%	6.33%	5.56%
	VRank	0.754	0.719	0.718	0.722	0.759	-	-	-	-	-
SR	Random	0.499	0.511	0.496	0.496	0.516	50.30%	40.31%	44.35%	45.16%	47.09%
	DeepGini	0.710	0.674	0.666	0.673	0.722	5.63%	6.38%	7.51%	6.98%	5.12%
	VanillaSM	0.709	0.677	0.668	0.672	0.726	5.78%	5.91%	7.19%	7.14%	4.55%
	PCS	0.704	0.672	0.666	0.670	0.725	6.53%	6.70%	7.51%	7.46%	4.69%
	Entropy	0.710	0.671	0.661	0.674	0.717	5.63%	6.86%	8.32%	6.82%	5.86%
	VRank	0.751	0.717	0.716	0.720	0.759	-	-	-	-	-
ZCA	Random	0.505	0.499	0.508	0.490	0.509	49.31%	44.09%	41.34%	47.35%	49.12%
	DeepGini	0.712	0.678	0.670	0.678	0.721	5.90%	6.05%	7.16%	6.49%	4.85%
	VanillaSM	0.711	0.681	0.672	0.677	0.724	6.05%	5.74%	6.85%	6.65%	4.42%
	PCS	0.706	0.676	0.671	0.674	0.725	6.80%	6.36%	7.00%	7.12%	4.28%
	Entropy	0.713	0.675	0.666	0.679	0.716	5.75%	6.52%	7.81%	6.33%	5.59%
	VRank	0.754	0.719	0.718	0.722	0.756	-	-	-	-	-
CSR	Random	0.495	0.499	0.493	0.498	0.518	48.28%	33.27%	39.76%	38.76%	46.53%
	DeepGini	0.691	0.617	0.634	0.634	0.725	6.38%	7.78%	8.68%	8.99%	4.55%
	VanillaSM	0.689	0.616	0.639	0.633	0.727	6.53%	7.95%	7.82%	9.16%	4.26%
	PCS	0.685	0.610	0.639	0.631	0.726	7.15%	9.02%	7.82%	9.51%	4.41%
	Entropy	0.691	0.615	0.627	0.635	0.718	6.38%	8.13%	9.89%	8.82%	5.57%
	VRank	0.734	0.665	0.689	0.691	0.758	-	-	-	-	-

better effectiveness over all the compared approaches on noisy video datasets.

Answer to RQ2: When applied to noisy datasets, VRank also demonstrates better effectiveness over the compared test prioritization approaches, with an average improvement of 4.26% to 53.56%. The improvement is consistently observed across each utilized noise generation technique.

3.5.3 RQ3: Impact of different ranking models

Objective: We explore the efficacy of various ranking models in VRank concerning their ability to leverage the generated video features for test prioritization.

Experimental design: In order to explore the influence of different ranking models

Table 3.7: Performance improvement of VRank on the 105 noisy subjects (i.e., 3(natural input sets)×5(Video classification models)×7(noise technique)

Approach	# Best cases	Average APFD	Improvement(%)
Random	0	0.499	38.68
DeepGini	0	0.645	7.29
VanillaSM	0	0.646	7.12
PCS	0	0.642	7.79
Entropy	0	0.644	7.45
VRank	105	0.692	-

Table 3.8: Effectiveness comparison on noisy datasets generated by the WS noise generation technique

Data	Model	Approach					
		Random	DeepGini	VanillaSM	PCS	Entropy	**VRank**
HWID12	C3D	0.478	0.698	0.699	0.690	0.696	0.719
	R2Plus1D	0.506	0.682	0.684	0.662	0.680	0.716
	R3D	0.503	0.667	0.669	0.667	0.661	0.715
	SlowFastNet	0.498	0.638	0.636	0.633	0.640	0.699
	VT	0.503	0.710	0.711	0.712	0.710	0.736
HMDB51	C3D	0.514	0.577	0.576	0.572	0.577	0.617
	R2Plus1D	0.504	0.555	0.558	0.554	0.551	0.599
	R3D	0.510	0.559	0.565	0.571	0.549	0.610
	SlowFastNet	0.499	0.551	0.551	0.549	0.551	0.577
	VT	0.497	0.659	0.659	0.650	0.658	0.701
UCF101	C3D	0.499	0.604	0.604	0.600	0.605	0.658
	R2Plus1D	0.504	0.621	0.620	0.616	0.620	0.670
	R3D	0.504	0.614	0.615	0.612	0.612	0.669
	SlowFastNet	0.499	0.583	0.581	0.578	0.585	0.630
	VT	0.496	0.711	0.712	0.706	0.708	0.756

on the effectiveness of VRank, we have proposed three VRank variants that employ different ranking models for the learning-to-rank process. Specifically, we evaluate VRank along with its variants, namely VRank^X, VRank^R, and VRank^L (as described in Section 3.3.5), on their ability to prioritize test inputs in both natural and noisy settings and assess their effectiveness in terms of APFD.

Results: The experimental results pertaining to RQ3 are presented in Table 3.9. The upper segment of the table displays the average effectiveness across diverse models, while the lower segment showcases the average effectiveness across different video datasets. We see that both VRank and its variants perform better than all the compared approaches on average. Specifically, in the case of natural inputs, VRank exhibits the highest performance in 86.67% of instances, whereas VRank^X achieves superiority in the remaining 13.33% of cases. In the context of noisy data, VRank achieves the highest performance in 84.61% of cases, while VRank^X excels in the remaining 15.38% of cases. Furthermore, the mean APFD values for VRank and its variants on the natural dataset range from 0.718 to 0.758, while the compared approaches exhibit mean APFD values ranging from 0.493 to 0.755. On the noisy dataset, the mean APFD values for VRank and its variants range from 0.680 to 0.746, while the compared methods exhibit mean APFD values ranging from 0.499 to 0.742. These findings demonstrate that VRank and its variants perform better than all the compared methods on average.

Furthermore, we see that VRank demonstrates the highest effectiveness among all variants. As shown in Table 3.9, regardless of whether the datasets are natural or noisy, VRank consistently achieves the highest average APFD across all cases. In the natural dataset scenario, the mean APFD of VRank reaches 0.734, while the variants exhibit mean values ranging from 0.701 to 0.729. In the noisy dataset scenario, VRank attains an average APFD of 0.702, whereas the variants exhibit mean values ranging from 0.672 to 0.697. These results indicate that VRank surpasses all variants in terms of effectiveness, suggesting that the ranking model employed by VRank, namely LightGBM, outperforms the ranking models utilized by the variants in leveraging the features of video input for test prioritization.

Table 3.9: Performance (APFD scores) of VRank variants with different ranking models
(#BC ⇔ #Best cases) and (Avg ⇔ Average APFD score)

Approach	Natural inputs						Noise inputs					
	#BC	C3D	R2Plus1D	R3D	SlowFastNet	VT	#BC	C3D	R2Plus1D	R3D	SlowFastNet	VT
Random	0	0.499	0.496	0.493	0.502	0.511	0	0.499	0.501	0.501	0.499	0.507
DeepGini	0	0.712	0.677	0.670	0.678	0.725	0	0.669	0.641	0.640	0.631	0.710
VanillaSM	0	0.710	0.679	0.672	0.676	0.727	0	0.668	0.642	0.643	0.630	0.712
PCS	0	0.706	0.675	0.670	0.673	0.727	0	0.664	0.636	0.642	0.627	0.711
Entropy	0	0.712	0.674	0.665	0.679	0.719	16	0.670	0.639	0.634	0.631	0.705
VRank^X	2	0.750	0.712	0.713	0.713	0.755	0	0.706	0.678	0.687	0.674	0.742
VRank^R	0	0.743	0.706	0.706	0.705	0.749	1	0.701	0.672	0.681	0.662	0.737
VRank^L	0	0.713	0.690	0.684	0.691	0.728	0	0.677	0.656	0.659	0.651	0.719
VRank	13	0.754	0.719	0.718	0.722	0.758	88	0.711	0.685	0.691	0.680	0.746

Approach	Natural inputs				Noise inputs			
	HWID12	HMDB51	UCF101	AVG	HWID12	HMDB51	UCF101	AVG
Random	0.508	0.492	0.501	0.501	0.502	0.501	0.502	0.502
DeepGini	0.706	0.634	0.737	0.692	0.685	0.607	0.683	0.658
VanillaSM	0.706	0.636	0.737	0.694	0.686	0.609	0.682	0.659
PCS	0.703	0.634	0.734	0.691	0.682	0.607	0.679	0.656
Entropy	0.705	0.629	0.735	0.690	0.683	0.603	0.681	0.655
VRank^X	0.739	0.664	0.783	0.729	0.725	0.639	0.728	0.697
VRank^R	0.731	0.661	0.773	0.722	0.715	0.635	0.721	0.690
VRank^L	0.714	0.636	0.753	0.701	0.699	0.614	0.704	0.672
VRank	0.742	0.672	0.789	0.734	0.727	0.645	0.735	0.702

Answer to RQ3: VRank and its variants exhibit better average effectiveness than DeepGini, Vanilla SM, PCS, and Entropy. Notably, VRank surpasses all of its variants in terms of effectiveness, indicating that the ranking model implemented in VRank, LightGBM, outperforms the ranking models employed by the variants in effectively utilizing the video input features for test prioritization.

3.5.4 RQ4: Feature contribution analysis

Objective: We aim to investigate the contributions of different types of features to the effectiveness of VRank.

Experimental design: To evaluate the importance of different types of features for VRank, we leverage the cover metric in the XGBoost algorithm [13]. The cover metric provides a means of evaluating feature importance by quantifying the average coverage of each instance through the leaf nodes within a decision tree. Specifically, this metric entails the calculation of the frequency with which a specific feature is employed for partitioning the data across all trees within the ensemble, followed by the summation of the coverage values associated with each feature across all trees. Subsequently, the resulting coverage value is appropriately normalized by the total number of instances, thereby yielding the average coverage of each instance by the leaf nodes. The significance of a particular feature is then ascertained based on its derived coverage value, with features exhibiting higher coverage values being attributed greater importance. Upon computing the importance scores for all features, the identification of the top-N important features was carried out for each dataset, thereby providing an elucidation of the feature types that significantly contribute to the effectiveness of VRank.

Moreover, in order to assess the impact of each feature type on the effectiveness of VRank, we conducted a carefully designed ablation study following the methodology outlined in prior research [107]. More specifically, we removed individual feature types

Table 3.10: Top-10 features in terms of the average contribution

Rank	HWID12		HMDB51		UCF101	
	Feature	Score	Feature	Score	Feature	Score
1	UF^{5}	88.56	UF^{0}	104.31	UF^{1}	280.18
2	UF^{1}	80.86	EF^{1352}	84.75	EF^{984}	248.26
3	PF^{11}	61.34	TF^{2070}	80.48	EF^{1386}	219.73
4	EF^{52}	38.23	EF^{443}	72.76	PF^{65}	211.38
5	EF^{2048}	37.49	UF^{2}	70.74	TF^{2153}	201.27
6	TF^{2104}	37.48	UF^{1}	68.38	UF^{5}	163.01
7	EF^{1456}	36.43	EF^{1819}	67.33	TF^{2164}	154.78
8	TF^{2113}	34.88	TF^{2115}	64.25	PF^{18}	147.84
9	TF^{2068}	33.43	EF^{2124}	64.06	TF^{2184}	146.92
10	PF^{7}	32.13	PF^{44}	63.45	UF^{5}	139.84

Table 3.11: Ablation study on different features of VRank: Embedding Features(EF), Temporal Features(TF), Prediction Features(PF), Uncertainty Features(UF). 'w/o' means 'without'

Approach	Dataset			Average
	HWID12	HMDB51	UCF101	
VRank w/o EF	0.727	0.656	0.772	0.718
VRank w/o TF	0.731	0.654	0.771	0.719
VRank w/o PF	0.729	0.653	0.768	0.717
VRank w/o UF	0.732	0.652	0.763	0.715
VRank	0.742	0.672	0.788	0.734

and evaluated VRank's effectiveness under these modified conditions. For example, to measure the contribution of UF features, VRank was executed with UF features excluded while retaining the other three feature types. The resulting performance of VRank was then evaluated under these adjusted circumstances. Similarly, to evaluate the contribution of EF features, VRank was executed without generating EF features while still generating the other three feature types. The performance of VRank was subsequently assessed in this context. Through the conducted ablation study, we can compare the contribution of each feature type to the overall effectiveness of VRank.

Results: The findings for RQ4 are presented in Table 3.10. In Table 3.10, the abbreviations UF, PF, EF, and TF represent uncertainty-based features, prediction features, video embedding features, and temporal features. Additionally, the small superscript numbers on the upper right corner of the feature abbreviations indicate the index of the feature. For instance, UF^{5} denotes the UF feature with index 5. In Table 3.10, we see that, across different video datasets (i.e., HWID12, HMDB51, and UCF101), all four types of features appear among the top 10 most contributing features. Specifically, for the HWID12 dataset, UF features contribute to 20% of the top 10 features, PF features contribute to 20%, EF features contribute to 30%, and TF features contribute to 30%. In the case of the HMDB51 dataset, UF, PF, EF, and TF features contribute 30%, 10%, 40%, and 20%, respectively. These experimental results illustrate that all four types of generated features make visible contributions to the effectiveness of VRank.

The experimental results of the ablation study are presented in Table 3.11. In this table, 'w/o' stands for 'without.' For example, 'VRank w/o EF' refers to executing VRank without generating the video embedding features. From Table 3.11, we see that the original VRank achieves the highest average effectiveness. Removing any

type of feature results in a decrease in the effectiveness of VRank, demonstrating that each type of feature contributes to VRank's effectiveness. For instance, on the HWID12 dataset, the average APFD value of the original VRank is 0.742. Removing video embedding features results in a decline of VRank's average APFD to 0.727, while the removal of temporal features causes a decrease to 0.731, prediction features to 0.729, and uncertainty features to 0.732.

From Table 3.11, we see that across different datasets, all four types of features contribute to VRank. Specifically, the average APFD decrease resulting from the removal of EF is 0.016. Removing TF leads to an average APFD decrease of 0.015, while PF removal results in an average APFD decrease of 0.017, and UF removal causes an average APFD decrease of 0.019. These differences are small. Moreover, taking the HMDB51 dataset as an example, the APFD decreases caused by removing the four types of features are 0.016, 0.018, 0.019, and 0.02, respectively. These experimental results suggest that all types of generated features contribute to the effectiveness of VRank.

Answer to RQ4: All four types of generated features, namely uncertainty features, prediction features, video embedding features, and temporal features, visibly contribute to the effectiveness of VRank.

3.5.5 Impact of the number of extracted frames on the effectiveness of VRank.

Objective: In VRank, two critical steps involve generating video embedding features (EF) and temporal features (TF) from the video-type test to predict the likelihood of the test being misclassified. To obtain EF and TF, we utilize established frame sampling techniques [68] to extract a fixed number of frames from the video-type test input. In this research question, we explore the impact of the number of extracted frames on the effectiveness of VRank. **Experimental design:** In the original VRank implementation, we extracted 16 frames during the generation of video embedding features and temporal features. To investigate the impact of the number of generated frames, we kept the other execution processes of VRank unchanged and only varied the number of frames extracted, specifically changing it to 4, 8, and 32 frames. The reason we chose these specific numbers of frames to extract is as follows: Selecting 4, 8, and 32 frames can cover a range of frame numbers from relatively low (4 frames) to relatively high (32 frames). This can assist researchers in understanding the impact of different frame count levels on the performance of VRank. We compared the effectiveness (measured by APFD) of VRank with the different number of frames generated. Through comparison, we aim to explore the impact of the number of extracted frames on the effectiveness of VRank.

Results: The results for RQ5 are presented in Table 3.12. Specifically, Frames-4 indicates that, during the video embedding feature and temporal feature generation step in VRank, four frames were extracted. Similarly, Frames-8, Frames-16, and Frames-32 correspond to the extraction of 8, 16, and 32 frames, respectively. From Table 3.12, we see that the effectiveness of VRank increases slightly with the number of extracted frames. In the HWID12 dataset, the effectiveness of VRank with 4 frames to 32 frames is as follows: 0.725, 0.735, 0.742, and 0.748. In the HMDB51 dataset, the effectiveness of VRank with 4 frames to 32 frames is 0.651, 0.663, 0.672, and 0.681. For the UCF101 dataset, the values are 0.752, 0.768, 0.789, and 0.795. We see that on each dataset, VRank's APFD values gradually improve with an

Table 3.12: Influence of the number of extracted frames on the effectiveness of VRank

Data	Frames-4	Frames-8	Frames-16	Frames-32
HWID12	0.724	0.735	0.742	0.748
HMDB51	0.651	0.663	0.672	0.681
UCF101	0.752	0.768	0.789	0.795

increase in the number of frames. However, the augmentation of frames affects the running time of VRank, impacting efficiency. The original VRank, which utilizes 16 extracted frames, has outperformed all the compared test prioritization methods, and the total execution time is only around 3 minutes. As shown in Table 3.3, the original VRank (16 frames) outperforms all the compared methods in all cases, with improvements ranging from 5.76% to 46.51%. Therefore, for a trade-off between efficiency and effectiveness, we select to extract 16 frames in the process of generating video embedding features and temporal features.

Answer to RQ5: The effectiveness of VRank increases with the number of extracted frames, but the improvement is slight.

3.6 Discussion

3.6.1 Limitations

[*Dependency on Visual Features*] One noteworthy limitation of the current implementation of VRank is its exclusive emphasis on extracting visual information from video data, neglecting the incorporation of speech or audio information. This singular focus on visual features hampers the comprehensive understanding of video content, as audio analysis plays a pivotal role in decoding the complete semantic meaning embedded within videos. The absence of audio analysis restricts the model's ability to capture important auditory cues, such as spoken dialogue, sound effects, or background music, which are integral components of video content. Consequently, the lack of audio analysis may impede the accuracy and effectiveness of the ranking process, as the model's comprehension of videos remains incomplete and insufficiently nuanced. To address this limitation, future iterations of VRank will include a robust audio analysis component, which will facilitate a more holistic and comprehensive approach to video ranking by encompassing both visual and auditory information. By incorporating audio analysis, VRank will be empowered to leverage the complementary nature of audio-visual data, enabling a more nuanced understanding of video content and enhancing the accuracy and reliability of the ranking process.

[*Contextual Understanding*] While VRank excels in the analysis of individual frames within a video, it can exhibit limitations in comprehending the broader contextual aspects and narrative structure inherent in video content. As focusing solely on individual frames, VRank can lack the temporal relationships and dependencies between frames, thus failing to capture the temporal dynamics and sequential nature of video content. This limitation can pose challenges in accurately ranking videos that heavily rely on temporal coherence, as well as those that require a comprehensive understanding of the entire video content as a cohesive narrative. The lack of contextual understanding may result in an incomplete representation of the video's meaning and can impact the effectiveness of the ranking process, particularly for videos with intricate storytelling or complex visual narratives. To mitigate

this limitation, future research efforts will seek to enhance VRank's contextual understanding capabilities by exploring methods that can capture narrative structures within videos. By incorporating contextual understanding, VRank will be better equipped to rank videos that exhibit nuanced temporal dynamics, thereby improving its overall performance and applicability in real-world scenarios.

[*Whole Video Classification*] Our research is centered around multi-class datasets that concentrate on classifying entire videos rather than categorizing each frame of a video individually. Specifically, in the video dataset we evaluated, each video (sample) is assigned to a specific category. This implies that within the evaluated video dataset, each frame belongs to the same category. For instance, in the UCF101 dataset, there are a total of 101 categories. A video sample classified as "High Jump" has each frame assigned to the "High Jump" category.

3.6.2 Threats to Validity

Threats to Internal Validity. The internal threats to validity primarily reside within the implementation of our proposed VRank framework and the test prioritization approaches utilized for comparison. To mitigate these threats, we took several measures to ensure the reliability and consistency of our experimental setup. Firstly, we implemented VRank using the widely recognized and extensively utilized PyTorch library, known for its robustness and computational efficiency in deep learning research. By leveraging a well-established framework, we aimed to minimize potential implementation biases. Furthermore, to guarantee the reliability of our comparative analysis, we employed the publicly available implementations of the compared approaches as provided by their respective authors. This approach ensures consistency across the experimental procedures, reducing the risk of implementation discrepancies and increasing the reproducibility of our findings. Another potential internal threat arises from the inherent randomness associated with the training process of the models. To mitigate this threat and enhance the stability of our experimental results, we conducted a statistical analysis. Specifically, we conducted multiple runs of all experiments, repeating the training and evaluation procedures ten times. By adopting this approach, our experimental findings acquire heightened reliability and stability. Furthermore, we calculated the statistical significance of our experimental results, providing further evidence for the validity and generalizability of our results.

Threats to External Validity. The external threats to validity in our study primarily stem from the generalizability of our findings to other models and video datasets. To address this concern, we carefully selected a diverse set of models and video datasets for our experimental evaluation. By incorporating various model-dataset pairs, we aimed to capture a broad spectrum of scenarios and ensure that our findings are not limited to a specific combination of models and datasets. We intentionally included both natural and noisy inputs during testing. More specifically, we leveraged well-established noise generation techniques from publicly available studies. These techniques, derived from the literature on image and video processing [93, 94, 95, 96], enable us to augment the diversity of the video datasets used for evaluation. By incorporating these augmentation techniques, we aimed to evaluate the effectiveness of VRank on noisy contexts.

3.7 Related Work

We present the related work in three aspects: test prioritization in DNN testing, deep neural network testing, and test prioritization for traditional software.

3.7.1 Test Prioritization in DNN Testing

In the domain of DNN testing, test prioritization [6, 9, 108, 47, 109] has emerged as a critical task for identifying possibly-misclassified test inputs. Various metrics have been proposed for this purpose. DeepGini, proposed by Feng *et al.* [6], aims to prioritize tests based on model uncertainty. DeepGini assumes that a test input is more likely to be mispredicted if the DNN outputs similar probabilities for each class. Byun *et al.* [110] evaluated several white-box metrics for prioritizing bug-revealing inputs, including widely-used metrics like softmax confidence, Bayesian uncertainty, and input surprise. Moreover, Weiss *et al.* [9] performed a comprehensive investigation of various DNN test input prioritization techniques, including several uncertainty-based metrics such as Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy. These metrics have been shown to be effective in identifying possibly-misclassified test inputs and aiding test prioritization efforts. Recently, Wang *et al.* [10] proposed PRIMA, which uses intelligent mutation analysis for prioritizing test inputs. This approach can be applied not only to classification but also to regression models and can handle test inputs generated from adversarial input generation approaches that increase the probability of the wrong class. While PRIMA has demonstrated its effectiveness on image data, it cannot be used to prioritize video tests since its mutation rules are not adapted to video data.

The aforementioned uncertainty-based test prioritization methods can be adapted for test prioritization of video datasets. However, video data possesses unique characteristics, particularly temporal information, which necessitate a tailored test prioritization strategy. In comparison to these existing approaches, our proposed VRank introduces a carefully-designed feature generation strategy specifically for video samples. VRank leverages frame sampling techniques [68] and the ResNet model [87] to extract frame representations that capture the temporal information embedded within video data. By exploiting these techniques, VRank enables the effective prioritization of video tests by considering the temporal dynamics and dependencies present in the video content, thereby augmenting the accuracy and relevance of the test prioritization process.

3.7.2 Deep Neural Network Testing

DNN Testing [60, 111] aims to systematically assess and enhance the reliability and robustness of neural network models through rigorous testing methodologies. In addition to test input prioritization, numerous approaches have been proposed to enhance the efficiency of DNN testing through the process of test selection. Test selection aims to accurately estimate the accuracy of the entire test set by labeling only a carefully chosen subset of test inputs, thereby reducing the labeling cost associated with DNN testing. By effectively selecting a representative subset of test inputs, test selection techniques can provide reliable estimates of the DNN's performance without requiring the evaluation of the entire test set. Li *et al.* [48] introduced two test selection methods, namely Cross Entropy-based Sampling (CES) and Confidence-based Stratified Sampling (CSS). CES operates by minimizing the cross-entropy between the selected set and the complete test set, thereby ensuring

that the distribution of the selected test set aligns with that of the original test set. In this way, CES aims to capture the diversity and characteristics of the complete test set while using only a fraction of the available test inputs. CSS, on the other hand, leverages the confidence features of test inputs to ensure similarity between the selected test set and the entire test set. By selecting samples based on their confidence scores, CSS aims to capture the representative distribution of the test set, thereby providing accurate estimations of the DNN's performance.

Building upon the foundation of test selection, Chen *et al.* [46] proposed a practical test selection approach called Practical Accuracy Estimation (PACE). PACE integrates various techniques, including clustering, prototype selection, and adaptive random testing, to facilitate efficient and effective test selection. PACE initiates by clustering all the test inputs based on their testing capabilities. Through this process, test inputs with similar characteristics and behaviors are grouped together, enabling the identification of distinct clusters within the test set. Following clustering, prototypes are selected from each cluster using the MMD-critic algorithm [49]. The MMD-critic algorithm ensures that the selected prototypes are representative of their corresponding clusters, thus capturing the diversity and variability of the test set. For test inputs that do not fall into any specific cluster, PACE employs adaptive random testing, which randomly selects samples from the remaining unclustered inputs. By adapting the sampling strategy to the unique characteristics of the unclustered inputs, adaptive random testing helps maintain the representativeness and diversity of the selected test set. It is important to note that while test selection techniques aim to reduce the labeling cost by selecting a subset of test inputs, our work primarily focuses on the complementary task of test prioritization. Unlike test selection methods that estimate the performance of a DNN by utilizing a selected subset of inputs, our proposed approach (VRank) focuses on ranking all test inputs based on their potential to reveal bugs without discarding any of them.

Besides enhancing the efficiency of DNN testing [8, 7, 20, 52, 112], evaluating the adequacy of DNNs has been a significant objective in several studies in the field. These studies have focused on developing metrics and frameworks to assess the coverage and effectiveness of test sets. Pei *et al.* [8] introduced the concept of neuron coverage as a metric to evaluate the adequacy of a test set in covering the logic of a DNN model. Neuron coverage measures the extent to which the activations of individual neurons in the model are exercised by the test inputs. Building upon this metric, the authors developed a white-box testing framework for DNNs, which has shown effectiveness in detecting faults and revealing hidden vulnerabilities in these models.

Ma *et al.* [7] proposed DeepGauge, a comprehensive set of DNN testing coverage criteria. One of the key components of DeepGauge is neuron coverage, which serves as a significant indicator of the effectiveness of a test input. By measuring the coverage of neurons in the model, DeepGauge provides insights into the regions of the model that are adequately exercised by the test inputs. Additionally, DeepGauge introduced new coverage metrics with varying granularities to differentiate between adversarial attacks and legitimate test data. These metrics capture the subtle differences in the behavior of the model when exposed to adversarial inputs, enabling the detection and identification of such attacks. Kim *et al.* [52] proposed the surprise adequacy approach for DNN testing. This approach assesses the effectiveness of a test input by quantifying its surprise with respect to the training set. The surprise of a test

input is measured by the difference in the activation values of neurons in response to the new input. By evaluating the surprise of test inputs, this approach provides a means to identify inputs that exhibit unusual or unexpected behavior, highlighting potential vulnerabilities or weaknesses in the model.

3.7.3 Test Prioritization for Traditional Software

Within the domain of software testing, various techniques have been explored and adopted to improve the efficiency and effectiveness of bug detection in the testing process [56, 57, 58, 113, 9]. Among these techniques, test prioritization has gained significant attention as a means to determine the most advantageous order in which to execute test cases, aiming to detect software bugs at the earliest possible stage.

The main objective of test case prioritization is to identify the maximum number of test cases that have the potential to reveal bugs within a limited time frame. Empirical studies have demonstrated the positive impact of test case prioritization on the fault detection rate of the overall test suite [102, 114, 115]. For instance, Di *et al.* [79] conducted a case study evaluating coverage-based prioritization strategies on real-world regression faults. Their study assessed the effectiveness of various test case prioritization techniques in detecting bugs, providing insights into the efficacy of different prioritization approaches. Rothermel *et al.* [116] introduced and compared three types of test case prioritization techniques for regression testing, which leveraged test execution information to determine the order of test case execution. Their research emphasized the effectiveness of these prioritization techniques in increasing the fault detection rate of the test suite. Lou *et al.* [56] proposed a test case prioritization approach based on the fault detection capability of individual test cases. They introduced two models, the statistics-based model, and the probability-based model, to calculate the fault detection capability of each test case. Through their empirical evaluation, they found that the statistics-based model outperformed other approaches, highlighting the importance of considering the fault detection capability in test case prioritization.

Shin *et al.* [57] developed a test case prioritization technique utilizing the diversity-aware mutation adequacy criterion. They empirically evaluated the effectiveness of mutation-based prioritization techniques using a large-scale collection of developer-written test cases. Their research shed light on the benefits of employing mutation-based prioritization techniques in practical testing scenarios. Papadakis *et al.* [58] proposed a method that involved mutating Combinatorial Interaction Testing models and prioritizing test cases based on their ability to detect and eliminate mutants. They demonstrated a strong correlation between the number of model-based mutants killed and the identification of code-level faults by the test cases, illustrating the potential of model-based prioritization approaches in software fault detection.

These studies collectively showcase the effectiveness and benefits of test prioritization techniques in detecting software faults and optimizing the overall software testing process. By strategically ordering the execution of test cases, testers can allocate their limited resources more efficiently and uncover bugs earlier, leading to improved software quality and reliability.

3.8 Conclusion

To solve the labeling-cost problem specifically in the context of video test inputs, we proposed a novel test prioritization approach called VRank. The primary objective

of VRank is to assign higher priority to video test inputs that are more likely to be misclassified. The fundamental concept underlying VRank is that test inputs situated closer to the decision boundary of the model are at a higher risk of being predicted incorrectly. To capture the spatial relationship between a video test and the decision boundary, we employ a vectorization technique that transforms a given video test into a lower-dimensional space to indirectly reveal the underlying proximity between the test and the decision boundary. To implement this vectorization strategy, we generate four different types of features for each video-type test: temporal features, video embedding features, prediction features, and uncertainty features. Each of these feature types captures essential aspects of the video tests and the model's classification behavior specific to videos. Temporal features capture the unique temporal coherence inherent in a given video-type test. Video embedding features encapsulate the inherent information within the videos, while the prediction features focus on the model's classification information regarding the videos. Uncertainty features, on the other hand, take into consideration the level of uncertainty associated with the model's classification outputs. By combining these feature types, VRank effectively constructs a comprehensive feature vector for each individual test input. To assess the misclassification likelihood of each test input, VRank employs a LightGBM-based ranking model that takes the constructed feature vector as input and generates a misclassification score. A higher misclassification score indicates a higher probability of the test input being incorrectly predicted by the model. Based on these misclassification scores, VRank sorts all the tests within the test set in descending order, establishing a prioritized ranking. To assess the effectiveness of VRank, we carried out an empirical evaluation, comparing it with several test prioritization methods. Our evaluations involved 120 subjects, incorporating both natural and noisy data. The results of our experiments demonstrate the effectiveness of VRank in comparison to a diverse range of existing test prioritization approaches. Specifically, VRank yielded an average improvement of 5.76%∼46.51% on natural datasets and 4.26%∼53.56% on noisy datasets.

4 Test Input Prioritization for Graph Neural Networks

In this chapter, we introduce a novel test prioritization approach called NodeRank, which focuses on prioritizing test inputs that are more likely to be misclassified by the evaluated GNN model. NodeRank addresses a critical gap in the literature: existing DNN prioritization methods ignore the interdependencies among test inputs (nodes) in graph-structured datasets. The core premise is that a test is considered more likely to be misclassified if it can kill many mutated models and produce different prediction results with many mutated inputs. By prioritizing such potentially misclassified test inputs, testers can allocate labeling resources more effectively and thus enhance debugging efficiency.

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

- Yinghua Li, Xueqi Dang, Weiguo Pian, Andrew Habib, Jacques Klein and Tegawendé F. Bissyandé. Test Input Prioritization for Graph Neural Networks. IEEE Transactions on Software Engineering (TSE). Accepted for publication on Mar. 31, 2024.

Contents

4.1 Introduction

Recent years have witnessed widespread adoption of graph machine learning for modeling, predictive, and analytics tasks on graph-structured data, while the emergence of Graph Neural Networks (GNNs) [117] has led to the achievement of the unprecedented performance of a variety of applications in drug design [118, 119, 120], recommender systems [121, 122], and social network analysis [123, 124]. As they are increasingly adopted, the debugging of GNNs becomes essential, especially in safety-critical and security-sensitive domains. A key perspective in that domain is developing effective and efficient techniques for GNN testing to achieve quality assurance.

Unfortunately, Deep Neural Networks (DNNs), including GNNs, are notoriously difficult to test due to the limitations in the availability of a test oracle [48, 6, 10]. Indeed, DNN testing is challenged by the fact that it is costly and time-consuming to label test inputs: 1) automated labeling is not yet mainstream; 2) datasets can be substantially large, and the data can be complex, as in the case of GNNs; 3) labeling may require deep domain-specific knowledge, which is prohibitively expensive to acquire. Therefore, to achieve efficient and effective testing of DNN-based systems, researchers and practitioners generally focus on identifying only the relevant test inputs that are likely to cause the system to behave incorrectly (i.e., bug-revealing test inputs). Diagnosing those inputs is then expected to provide insights for debugging the DNNs.

Prior work has developed various techniques to identify and prioritize bug-revealing test inputs, which allows testers/developers to focus on the most critical inputs [6, 10, 125, 110]. Such test prioritization techniques aim at optimizing the time as well as the required resources for testing. A large majority of DNN test prioritization approaches fall within three categories [10]: coverage-based, confidence-based and surprise-based approaches. Confidence-based approaches, such as DeepGini [6], prioritize test inputs based on model confidence: a test input is more likely to be incorrectly predicted via a DNN model if that model outputs similar prediction probabilities for each class. Coverage-based approaches, such as CTM [11], simply adapt coverage-based test prioritization from traditional software systems testing into DNN testing and have been shown to underperform against confidence-based approaches [6]. Surprise-based methods [52, 110] perform test prioritization based on the surprise of test inputs. This "surprise" is quantified by measuring the distance in neuron activation patterns between a test input and the training data. However, existing studies [105] have demonstrated that surprise-based methods are less effective than confidence-based approaches. Furthermore, surprise-based methods typically come with higher computational costs due to the need for more parameter tuning.

Although confidence-based approaches have demonstrated effectiveness in the context of DNNs, they suffer from several limitations when applied to GNNs. Notably, they do not account for the interdependence inherent in graph-structured test inputs composed of nodes and edges. These approaches were originally designed for DNNs, where tests are independent of each other. Additionally, confidence-based approaches operate under the assumption that test inputs for which the model exhibits low confidence are more likely to be misclassified and, therefore, should be given higher priority. However, in the presence of adversarial attacks, the model's confidence can be higher for incorrect predictions, leading to erroneous outputs.

More recently, novel approaches such as PRIMA [10] are being introduced in the literature of DNN testing, leveraging techniques such as mutation analysis. However, PRIMA, the state-of-the-art in DNN test prioritization, cannot be applied to GNNs since their mutation operators are not adapted to graph-structured data and models. Dang *et al.* [108] proposed GraphPrior, a test prioritization method specifically designed for GNNs. Despite GraphPrior also relying on mutation analysis, there are significant differences between NodeRank and GraphPrior:

- **Incorporating Input Mutations** GraphPrior performs test prioritization solely based on model-specific mutations, whereas NodeRank not only considers model mutations but also takes into account mutations specific to the input. NodeRank considers two types of input mutations: 1) Node feature mutations, which are designed to perturb the feature attributes of selected nodes, consequently influencing the representation and information flow within the graph; 2) Graph structure mutations, which aim to alter the interdependence of the test inputs within the graph by introducing additional edges, thus changing the structural properties of the graph.
- **Leveraging Ensemble Learning Techniques for learning-to-rank** In contrast to GraphPrior, which employs a single ranking model to learn the misclassification probability of test inputs, NodeRank leverages ensemble learning techniques to integrate multiple base ranking models with the aim of optimizing its performance. Existing studies [126, 127, 128] have demonstrated that ensemble learning typically achieves higher accuracy than single ML models. Furthermore, our analysis delves into the influence of different ensemble techniques on NodeRank and illustrates that the sum-based ensemble technique yields the best performance.
- **Considering Different Killing Methods** GraphPrior simply assumes that a mutated model is considered "killed" if the predictions of the original model and the mutated model for the test input differ. However, prior research [60] has highlighted that in the context of DNN mutation analysis, variations in the outputs between a mutated model and the original model can occur solely due to the inherent randomness in the training process rather than because the mutant is actually discriminated from the original model. Therefore, we utilized the killing method provided by DeepCrime [60] for test prioritization and generated relevant variants of NodeRank. In DeepCrime, the killing process involves iteratively training both the original model and the mutated model, then comparing the distribution difference in their outputs to determine whether the mutated model is "killed." This approach can contribute to mitigating the impact of randomness in the training process. Based on the DeepCrime approach, by comparing NodeRank variants utilizing model mutation rules and those not utilizing model mutation rules, we demonstrated that mutations generated by the model mutation rules of NodeRank contribute to its effectiveness.

This paper. We propose NodeRank (**Node Rank**ing for graph-structure test inputs), a novel test input prioritization approach targeting GNNs. NodeRank leverages the ideas from traditional mutation testing [129, 130] to prioritize potentially misclassified test inputs so that such tests can be identified earlier with limited manual labeling costs. More specifically, the core idea of NodeRank is that: a test is considered more likely to be misclassified if this test can kill many mutated models and produce different prediction results with many mutated inputs.

NodeRank is a test prioritization approach that is model-based, input-based, and

mutation testing-based. It applies mutation operations to GNN models and tests, generating mutation features for test prioritization. Specific mutation operations applied are described below.

In NodeRank, we developed three distinct types of mutations, namely graph structure mutation (GSM), node feature mutation (NFM), and GNN model mutation (GMM), based on the characteristics of GNNs and the graph test dataset. GSM aims to modify the interdependence of the graph test inputs by introducing additional edges, thereby altering the structural properties of the graph. NFM, on the other hand, perturbs the feature attributes of selected nodes, thereby influencing the representation and information flow within the graph. Both GSM and NFM can be categorized as input mutations as they directly modify the characteristics of the dataset. In contrast, GMM is specifically developed to mutate GNN models, with the objective of modifying the message passing of the GNNs by changing specific training parameters. The GMM mutation type thus falls under the category of model mutations.

For each test input, NodeRank generates these three types of mutations, as described above. Subsequently, by comparing the prediction results before and after the mutation, NodeRank generates a mutation feature vector for each test. Specifically, for graph input mutation (i.e., GSM and NFM), if a mutated input fails (i.e., the predictions for the mutated inputs and the original inputs are different), the corresponding element in the relative feature vector is marked as 1; otherwise, it is marked as 0. For GNN model mutation (i.e., GMM), if a mutated model is killed (i.e., the prediction results for this input via the mutated models and the original models are different), the corresponding element in the relative feature vector is marked as 1; otherwise, it is marked as 0. The mutation feature vector of each test is then fed into pre-trained ranking models, which are designed to predict the likelihood of this input being misclassified. Our ranking models are trained to automatically predict a misclassification score indicating its likelihood of being misclassified by the model.

To further enhance the performance of our ranking models, we adopt ensemble learning techniques that combine the predictions from multiple base ranking models. The idea draws inspiration from the field of ensemble learning [126, 127, 131], which aims to improve overall performance by integrating predictions from two or more base machine learning models. Notably, ensemble learning methods have achieved state-of-the-art outcomes across various machine learning applications [132, 133, 134, 135]. In the NodeRank framework, we employ three distinct ensemble methods [126, 136, 137] to effectively combine the outputs of the individual ranking models.

It is important to note that, NodeRank differs from the state-of-the-art test prioritization approach, PRIMA, in several domains: the **target** (GNN vs. DNN) as well as the **approach** (mutation rules and ranking strategies).

- **Target.** NodeRank is designed to address the test prioritization problem in GNNs and, therefore, operates on datasets that exhibit complex interdependence between individual test inputs. In contrast, PRIMA is intended for traditional DNNs, where each sample in the dataset is independent.
- **Mutation rules.** NodeRank's mutation rules can affect the interdependency between test inputs from two perspectives: First, NodeRank's model mutation rules can directly or indirectly affect the message passing between nodes in graph data. 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. Second, NodeRank's node mutation rules modify the interdependence between nodes by adding edges to nodes. When adding a new edge from node A to node B, a new connection is built, and the prediction of node A is now impacted by the newly connected node B, thus changing the node interdependence. In contrast, the mutation rules of PRIMA are specifically designed for independent test inputs and, therefore, do not impact the relationships between tests.

- **Ranking strategies.** NodeRank leverages ensemble ranking models to learn from mutation results for test prioritization. These models are constructed by combining different base ranking models, thereby improving the overall performance of the model [126, 127, 131]. In contrast, PRIMA employs a single ranking model for test prioritization.

We evaluate the performance of NodeRank based on 124 subjects (i.e., a pair of dataset and GNN model). Our evaluation considers both natural inputs and graph adversarial inputs, which are generated by eight graph adversarial attacks [138, 139, 140, 141]. We compare NodeRank with multiple test prioritization approaches. Our experimental results demonstrate that, on natural datasets, the average improvement of NodeRank over the compared approaches, in terms of APFD, is between 4.41% and 58.11%. On graph adversarial inputs, the average improvement of NodeRank over the compared approaches in terms of APFD ranges from 4.96% and 62.15%.

NodeRank can be applied across diverse real-world contexts. For instance, a typical use case of node classification in GNNs is fraud detection [142] in banking transfer transaction systems. Here, each account can be represented as a node, while the transactional interactions between them can be represented as edges. Through node classification, these accounts can be categorized as normal or fraudulent. When developers use GNNs to predict whether each node (account) is a fraudulent account, the GNNs can exhibit wrong prediction behavior, such as predicting fraudulent accounts as normal accounts, which can lead to losses for the bank. In this scenario, NodeRank can be utilized to prioritize potentially misclassified accounts (with those more likely to be misclassified ranked at the top). These sorted accounts can be provided to bank staff, allowing them to quickly perform manual checks on the accounts that are more likely to be misclassified, thus reducing losses.

The contributions of this paper are as follows:

- **Approach.** We propose a novel approach, NodeRank, to prioritize test inputs for GNN models. NodeRank introduces three distinct types of mutation rules that target the mutation of graph structure, node features, and GNN models, respectively and adopt ensemble-learning-based learning-to-rank to intelligently combine mutation results for effective test input prioritization.
- **Study.** We conducted a large-scale study based on 124 subjects to evaluate the effectiveness of NodeRank on both natural and adversarial inputs. The experimental results demonstrate its effectiveness.
- **Performance Analysis.** We provide an extensive analysis of the performance of NodeRank by investigating the influence of the different ensemble learning strategies as well as by performing an ablation study to showcase the contributions of the different mutation feature sets.

Our dataset, code, and results are made publicly available in a replication package[1] for the community.

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

4.2 Background

We now briefly introduce the key domain concepts for our work.

4.2.1 Graph Neural Networks

Graph neural networks [25, 26, 27] have achieved great success in solving machine learning problems on graph-structured data [123, 143, 144]. Initial models learned representations of target nodes by propagating neighborhood information through recurrent neural architectures in an iterative manner until a stable fixed point is reached. Subsequently, several variations have been proposed in the literature: Kipf *et al.* [145] proposed Graph convolutional networks (GCN), which adapt convolution techniques from classical convolutional neural networks, to graph data. GCN implements message passing of multi-order neighborhoods by superimposing several convolutional layers. More recently, other GNN architectures have been proposed towards taking into account the advancements in the field of deep learning: for example, Veličković *et al.* [27] proposed graph attention networks (GAT) which uses attention techniques to assign different weights according to the importance of nodes in the graph.

In GNNs, a graph is usually defined as a data structure composed of nodes and edges. We denote a graph as $G = (V, E)$ where $V = \{1, 2, \ldots, N\}$ refers to the set of N nodes, and $E \subseteq V \times V$ refers to the set of edges. In GNN datasets like Cora (a node classification dataset), each node represents a scientific paper, while edges represent citation relationships between papers. In this dataset, test inputs typically refer to new nodes (scientific papers) that have not been seen during the training process. In the case of the Cora dataset, given a test input (a scientific paper), a GNN model is used to classify the paper into specific categories. In other words, the GNN model predicts the categories that best describe the content of the given paper. For instance, these categories can be "reinforcement learning" and "neural networks," implying that the paper belongs to the "reinforcement learning" or "neural networks" category.

[**GNN training process**] GNNs undergo a training process similar to other neural networks. The inputs required for GNN training typically include: 1) **Graph Structure.** Graph structure information encompasses the connections between nodes in the graph.; 2) **Node Features.** Each node typically comes with associated feature vectors, which reflect the attributes of the node; 3) **Target Labels.** In the training data for GNN node classification, "Target Labels" refer to the category to which each node belongs. These labels are typically predefined.

During the training process of GNNs, several components are continually trained and optimized: 1) **Model Parameters.** The primary aim of GNN training is to refine the model parameters. These parameters include weights and biases linked to operations like graph convolutions and aggregation functions within the GNN architecture. 2) **Node Embeddings.** GNNs comprise layers with associated parameters, and part of the training process involves learning these node embeddings. Node embeddings are vector representations of individual nodes within the graph. They capture a node's structural and feature-based information and evolve as the model trains; 3) **Loss Function.** The loss function plays a pivotal role in GNN training. It quantifies the disparity between the model's predictions, typically pertaining to nodes or graph-level attributes, and the actual ground truth labels for the given task. Throughout training, model parameters are iteratively adjusted to

minimize this loss function.

The specific training process for GNNs typically consists of the following steps:

- **Initialization**: All GNN parameters are randomly initialized, typically with small random values.
- **Forward Propagation**: For each node, its node embedding is updated based on the information from its neighbors. This is typically achieved using weight matrices and aggregation functions (e.g., mean or max pooling). This aggregation process can go through multiple layers, allowing information to propagate further in the graph.
- **Loss Calculation**: This process calculates the loss value based on the GNN's output and the true labels.
- **Backpropagation**: This process computes the gradients of the loss function with respect to each parameter.
- **Parameter Weight Updates**: This process updates each parameter weight based on the gradient values.
- **Iterate**: This process repeats the above steps (forward propagation, loss calculation, backpropagation, parameter updates) until a stopping condition is met, such as reaching a predefined number of iterations.

It is important to note that graphs used in GNN training differ from normal graphs. Specifically, GNN training graphs include feature attributes for nodes. Furthermore, in tasks like node classification, nodes in the graphs have category labels. In contrast, normal graphs usually comprise only the topological structure of nodes and edges, without specific labels or node attributes.

[**GNN inference**] In the context of GNNs, *inference* refers to using a pre-trained GNN model to perform prediction on new graph data. For example, in node classification tasks, the GNN model utilizes its learned parameters and weights to classify a given node. The input typically consists of the features of the node and the graph structure of its belonged graph. The output is the classification of the node.

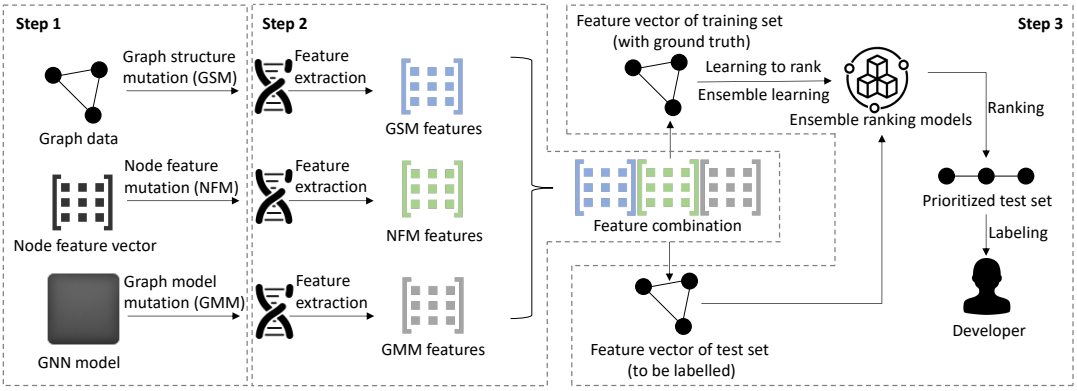


Figure 4.1: Overview of NodeRank

4.2.2 Mutation Testing

Mutation testing [129] is a software testing method that aims to evaluate the quality of the test suite by intentionally introducing small changes (called mutations) into the source code and observing the test suite's reaction. The core objective is to determine the effectiveness of test suites in finding code bugs. The intuition is that if a test case can detect intentionally-introduced errors, this test case is more likely to detect real bugs in practice. Mutation testing has achieved state-of-the-art performance by providing a comprehensive evaluation of the test suite via creating

and testing multiple variations (mutants) of the code, ensuring that the test cases are thoroughly covering different scenarios and even edge cases, which is difficult to achieve through traditional testing methods.

In mutation testing, *kill* and *fail* are terms used to describe the results of running a test suite on a set of artificially created code changes or 'mutants' to evaluate the quality of the test suite. Specifically, a mutant is regarded as 'killed' if the behavior of this mutant differs from that of the original code, indicating that the test suite is capable of detecting the fault introduced by the mutant. A test input is said to 'fail' if it is not passed by the target program.

4.2.3 Ensemble Learning

Ensemble learning [126] is a meta approach in machine learning where multiple, generally diversified, ML models are combined to achieve better performance and generalization. Common examples of ensemble learning strategies include Majority voting and Stacking. Majority voting is a straightforward approach that sums for each prediction class the number of yielded predictions by the different models: the class with the majority number of predictions is then outputted by the ensemble model. On the other hand, Stacking utilizes a meta-model such as logistic regression to learn how to optimally combine the predictions from base ML models.

Ranking is crucial to many real-world applications, notably in the field of information retrieval. In software engineering, test prioritization assumes the possibility to rank test cases according to their ability to reveal faults. In recent studies [126], the ranking has been formalized as a machine learning problem, and *ensemble ranking* often employs ensemble learning techniques to learn optimal weights for combining multiple ranking algorithms.

4.3 Approach

4.3.1 Overview

NodeRank is a model-based, input-based, and mutation testing-based test prioritization approach. By employing mutation operations on both GNN models and test inputs, NodeRank produces mutation features for each test input and predicts the misclassification probability of the input in order to perform test prioritization. Figure 4.1 presents the overview of the different steps in our NodeRank test prioritization approach. First, we offer detailed explanations for certain elements in Figure 4.1 and provide reasons for the symbols utilized for them, with the goal of enhancing the understanding of the figure.

- The second dotted square in Step 1 represents an $N \times M$ matrix. This matrix is used to encapsulate the feature vectors of all nodes. Specifically, each row of the matrix represents a node's feature vector. There are N rows in the matrix corresponding to the N nodes in the dataset. M columns represent that each node has M features.
- The reason we use a dotted representation for node feature vectors and mutation features is that, in our experiments, these features are both represented using matrices. The dotted square can serve as a visual abstraction of a matrix. A matrix comprises multiple values, and we use dots to represent the values within the matrix abstractly. For example, in the second dotted square in Step 1, the third dot in the second row represents the value of the third feature of the second

node in the graph dataset.

- It is important to note that the meaning of the dotted squares in Step 1 and Step 2 is different. However, since they both represent matrices, we use dotted squares with different colors to distinguish them. In Step 1, the dotted square represents the node feature vector, while in Step 2, the dotted square represents mutation features generated from the mutation results.
- We utilize arrows to illustrate processes and operations. For instance, in Step 1, the Graph data undergoes graph structure mutation within Step 1 and feature generation in Step 2, resulting in graph input mutation features. Another example involves the Node feature vector in Step 1, which undergoes node feature mutation in Step 1 and feature generation in Step 2 to yield node mutation features.
- The chromosome symbols represent mutation results. In Figure 4.1, a chromosome with a break indicates the mutations applied to the GNN model or inputs, resulting in mutation results. These mutation results are subsequently used for mutation feature generation in order to perform test prioritization.

Moreover, each box represents a step in the NodeRank workflow. In the following section, we offer a general overview of each step, as depicted in Figure 4.1. Specific details for each step can be found from Section 4.3.2 to Section 4.3.4.

❶ *Generating mutants.* NodeRank generates mutants for three different inputs: the graph structure itself (which represents the interdependence of samples in the datasets), the node features (which represents sample data), and the GNN model (which is learned and is the target of testing). To that end, we develop specific mutation rules that are carefully designed for GNN testing. Section 4.3.2 details those rules that must be applied to generate mutants for a given test set T, the graph structure G of the data, and the GNN model M under test.

❷ *Extracting and combining mutation features* NodeRank then obtains the model prediction towards the mutants and the original test inputs. By comparing the predictions, NodeRank generates the mutation feature vector for each input. The detailed description is as follows. Given M', a mutant of M, NodeRank considers that a test input kills M' if the prediction on this test input by M' is different from the prediction by M. For a given mutant of a test input $t \in T$, NodeRank considers that this mutant failed if it leads to a prediction that is different from the prediction using t. Given G', a mutant of G, NodeRank considers that the mutant fails if the prediction of the GNN using G' is different from its prediction when using G.

Based on the execution outputs, NodeRank builds feature vectors to train a ranking model. These are referred to as **mutation features** and are of three types: Node mutation features, graph structure mutation features, and model mutation features (cf. Section 4.3.3 for details).

❸ *Ranking test inputs using ensemble ranking models.* Eventually, for each test input, NodeRank produces three vectors, which represent three types of mutation features. These vectors are then concatenated to produce a mutation feature vector v for each test input $t \in T$. Given all test inputs from T, NodeRank, therefore, leverages ensemble ranking models based on their associated mutation features to predict ranking scores of the test inputs. These scores, ordered in a descending way, are used to prioritize the associated test inputs accordingly.

The findings presented in Section 4.5 provide compelling evidence for the effectiveness of NodeRank, which can be attributed, in part, to the careful design of mutation rules and the effective ensemble strategy of ranking models. 1) Our

designed mutation rules can effectively generate informative mutation features by leveraging the interdependence of test inputs. The node mutation rules operate by introducing new edges between nodes in the graph dataset, which impacts the interdependence structure of the data. The model mutation rules affect message passing between nodes in the GNN prediction process, leading to small changes in node interdependence. 2) NodeRank adopts an ensemble ranking model for test prioritization, which leverages the strengths of multiple base ranking models to improve the overall performance. By comparing different ensemble strategies, we are able to identify the most suitable approach for use in NodeRank's test prioritization process.

In the remainder of the section, we will describe in detail the mutation rules that we have designed for NodeRank (cf. Section 4.3.2), the construction process of the mutation feature vectors (cf. Section 4.3.3), the setup of the ensemble ranking model (cf. section 4.3.4) and the application of NodeRank (cf. Section 4.3.5).

4.3.2 Specifying Mutation Rules

We design mutation rules that are adapted to the key main ingredients of a GNN: the graph structure of the data, the nodes in the graph, and the GNN model itself, which are explained in detail as follows.

4.3.2.1 Graph structure mutation (GSM)

Graph structure mutation is designed to introduce slight changes to the input graph by randomly incorporating new edges. Consequently, when provided with a test input node, denoted as $t \in T$, we create mutants by adding one or more edges between node t and a randomly selected node, denoted as $s \in T$. For a given node $t \in T$, the following mathematical formula provides an intuitive representation of the GSM mutation:

$$G' = G + \sum_{i=1}^{n} \text{addEdge}\,(G, t, s_i) \tag{4.1}$$

where G represents the original graph. G' represents the mutated graph structure. In each iteration, we use the *addEdge* function to generate an edge from node t to a randomly selected node $s_i \in T$. We use the symbol "+" to denote the addition of the newly generated edge to the original graph G. This process is repeated n times, resulting in the addition of n edges to the original graph G.

4.3.2.2 Node feature mutation (NFM)

Given a test set and the features of the test inputs, node feature mutation aims to slightly change the features of the targeted nodes in order to offset their position in the feature space. This offset implies the modification of feature values in the different dimensions.

In the following, we introduce how node feature mutation is performed in detail. Given the original test set T, which consists of n nodes, each node t is characterized by m dimensions, where each dimension corresponds to a specific feature value of the node n. In this case, T can be represented as an $n \times m$ feature matrix. To perform node feature mutation, we apply an offset to this matrix. Specifically, assuming the degree of offset is denoted as α, Formula 4.2 represents the mutation process for the test set T. As observed in the formula, the initial step involves multiplying the matrix of the original test set T by the offset degree to calculate the ultimate offset

to be applied to T. Subsequently, the matrix of the mutated test set, denoted as $F(T')$, is derived by adding T's matrix to the offset $\alpha * F(T)$.

$$F(T') = F(T) + \alpha * F(T) \tag{4.2}$$

where $F(T')$ is the feature matrix of the mutated test set T', $F(T)$ is the feature matrix of the original test set T, and α is the coefficient of the degree of offset.

4.3.2.3 GNN model mutation (GMM)

Given a trained graph neural network model, the GNN model mutation aims to change the training parameters slightly. Formula 4.3 offers an intuitive representation of the GMM mutation. For integer or float type parameters, the mutation operation involves making slight adjustments to the parameters. In the case of Boolean-type parameters, the mutation operation involves switching between True and False. Therefore, the formula is as follows:

$$M' = \begin{cases} M(\theta + \beta \cdot \theta) & \text{if } \theta \in \mathbb{R} \\ M(\neg\theta) & \text{if } \theta \in \{ \text{ True, False } \} \end{cases} \tag{4.3}$$

where M' refers to the mutated GNN model. M refers to the original GNN model, θ refers to a parameter of the original model M, and β refers to the coefficient of change, indicating the magnitude of parameter change. The symbol $\neg$ signifies the logical negation operation, which inverts the parameter θ. If the original value is True, it becomes False, and if the original value is False, it becomes True.

In NodeRank, we consider the following:

- **Learning Additive Bias (LAB)** [145, 146, 147] The LAB parameter is a Boolean variable that determines whether to introduce a predetermined offset to the representation vectors of nodes in the GNN model. By enabling the LAB parameter (set to True), a bias parameter is assigned to each node's representation vector. This allows the GNN model to capture the intrinsic properties of the graph better and improve the interdependence between nodes in the prediction process.
- **Negative Slope (NS)** [146] NS is a float parameter that controls the slope of the negative part of the activation function used in the Gated Linear Unit (GLU) operation, a commonly used non-linear function for message passing in GNNs. In particular, GLU combines 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 of the activation function in the GLU operation determines the rate of decrease for negative input values and can affect the message passing between nodes. As such, the value of NS plays a crucial role in determining the sensitivity of the GNN model to negative input values and the resulting impact on the interdependence between nodes in the graph.
- **Changing Multi-head Attentions (CMA)** [146] CMA is an integer type parameter that determines the number of attention heads employed by the GNN model, with an increase in CMA leading to an expanded model capacity and improved capacity to capture the interdependencies that exist among the nodes in the graph.
- **Concat (CON)** [146] The CON parameter is a Boolean-type parameter that determines the method used to integrate node embeddings from neighboring nodes.

When set to True, the concatenation operation is employed to combine the node embeddings of adjacent nodes, resulting in a more sophisticated and expressive node representation. This, in turn, enhances the capacity of the GNN model to capture more interdependencies between nodes.

- **Adding Self Loops (ASL)** [145, 146] The ASL parameter is a Boolean parameter that governs the addition of self-loops to the input graph in graph neural networks. By setting ASL to 'True', self-loops are introduced to each node in the graph, enabling the aggregation of intrinsic information from nodes into their representation vectors. This operation modifies the weighting of neighboring nodes and can affect the interdependence of nodes during the prediction process.
- **Adding Layer Computations (ALC)** [145] ALC is a Boolean type parameter that determines whether or not to include additional layers of computation in the GNNs. When ALC is set to true, additional layers are introduced to the network, which allows for more complex transformations of the node features. As a result, the message passing process becomes more refined and capable of capturing more intricate dependencies among the nodes.
- **Hidden Channel (HC)** [145, 146, 147, 148] The HC parameter is an integer configuration parameter that governs the dimensionality of the hidden representation in each layer of the GNNs. As such, modifications to this parameter can impact the interdependence of nodes in a given graph by allowing the GNN to learn more expressive and informative node embeddings.

We explain how the mutation rules of NodeRank utilize node interdependence to generate mutations as follows:

- For Model-level mutants: NodeRank's mutant rules can directly or indirectly affect the message passing between nodes in graph data. 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.
- For Node-level mutants: NodeRank modifies the interdependence between nodes by adding edges to nodes. When adding a new edge from node A to node B, a new connection is built, and the prediction of node A is now impacted by the newly connected node B, thus changing the node interdependency.

Note that the mutation rules of NodeRank are specifically developed for GNNs, and its applicability in the context of DNNs has not yet been examined. Specifically, regarding node mutation rules, NodeRank focuses on modifying the connection relationships between nodes in a graph. However, in DNNs, the samples within a dataset are independent and lack any inherent connectivity, rendering the proposed mutation rules unsuitable for such datasets. Moreover, the model mutation rules of NodeRank are designed to impact the message passing between nodes during the prediction process, either directly or indirectly. In contrast, conventional DNNs generally consist of independent samples within a dataset, implying that such mutation rules are unlikely to influence the transmission of information between distinct tests.

4.3.3 Constructing Mutation Features Vectors

Leveraging the three types of mutation rules introduced in the previous steps, we generate a mutation feature vector for each test input. To this end, we execute the three mutation rules, thereby generating three distinct feature vectors for each input. These feature vectors are concatenated to build the final mutation feature vector. In

the following, we explain the generation of each feature vector of different mutation types.

Dataset mutation (NFM and GSM) Given a test input t and a GNN model M, we denote the mutants of t as $\{t_1, t_2, \ldots, t_n\}$, which are obtained using NFM mutation rules. We associate a vector V of size n to the test input t where n is the number of mutants and $V[k]$ maps to the execution output for the mutant t_k. *If t_k fails (i.e., the prediction of t_k is different from that of t), then $V[k]$ is set to 1. Otherwise, it is set to 0.* We use the same procedure to build a graph mutation features vector for t using the GSM mutation rules. Formula 6.14 describes the process of dataset mutation in our mutation testing operation.

$$V[k] = \begin{cases} 1 & \text{if } M(t_k) \neq M(t) \\ 0 & \text{if } M(t_k) = M(t) \end{cases} \tag{4.4}$$

where $M(t_k)$ represents the prediction of the GNN model M for mutant t_k, and $M(t)$ represents the prediction for the original test input t.

Model mutation (GMM): Given a test input t, a GNN model M and its mutants $\{M_1, M_2, \ldots, M_n\}$, we associate to the test input t, a vector V of size n (i.e., the number of mutants of M) where $V[k]$ maps to the execution output for the mutant M_k with test input t. *If t kills the mutated model M_k (i.e., the prediction of v_k via the original model M and the mutated model M_k is different), then $V[k]$ is set to 1. Otherwise, it is set to 0.* Formula 4.5 presents the process of model mutation in our mutation testing operation.

$$V[k] = \begin{cases} 1, & \text{if } M(t) \neq M_k(t) \\ 0, & \text{if } M(t) = M_k(t) \end{cases} \tag{4.5}$$

where $M(t)$ represents the prediction of the original model M for the test input t. $M_k(t)$ represents the prediction of the mutant model M_k for the same test input t.

4.3.4 Building an Ensemble Ranking Model

Based on the previous step, NodeRank generates a feature vector V_i for each test input $t_i \in T$. This feature vector is then used as the input to the ensemble ranking model for predicting the misclassification probability of t_i. The design of the ensemble ranking models is motivated by the principles of learning-to-rank [149] and ensemble learning [126]. In particular, we adopt four base ranking models, including Logistic Regression [150], Random Forest [90], XGBoost [13], and LightGBM [89], to form ensemble models that can leverage the strengths of each individual model. NodeRank uses the sum-based ensemble learning method [126], which combines scores of the base ranking models for a given test input. By inputting V_i into the sum-based ensemble ranking model, NodeRank obtains a misclassification score for t_i, which can be used to estimate the probability that the GNN model M will misclassify t_i.

Our experiments further consider two other ensemble learning methods (i.e., stacking-based [136] and voting-based [126]) to build variants of NodeRanks and assess the effectiveness of our design choices (cf. Section 4.4.6).

4.3.5 Usage of NodeRank

The inputs of NodeRank are a test set T and a GNN model M. The output is the prioritized test set T^P. NodeRank generates mutants for the test set T and the GNN model M and exploits the execution outputs of the GNN on these mutants to build feature vectors that can be utilized to learn to prioritize test inputs using ensemble ranking models. We present the training process of each ranking model as follows.

❶ **Dataset Split** Given a GNN model M with dataset T, we partition the dataset T into two subsets: a training set R and a test set. Following common practice in the field [92], we allocate 70% of the data to the training set and consider the remaining 30% as the test set. We emphasize that the test set is kept entirely separate from the training process and is only utilized to evaluate NodeRank.

❷ **Training set construction** Based on the given training set R, the objective of this step is to build a training set R' for training the ranking models. Firstly, for each input $r_i \in R$, three types of mutants are generated, and based on the execution of these mutants, the mutation feature vector V_i of r_i is obtained. Subsequently, the mutation feature vector of r_i is utilized to build the features of the training set R'. Secondly, the original GNN model M is used to classify each input $r_i \in R$ and compare it with the ground truth of r_i. This step helps identify whether r_i is misclassified by the GNN model M. If r_i is misclassified by M, it is labeled as 1, and if not, it is labeled as 0. This process aids in building the labels of the ranking model training set R'.

❸ **Training ranking models** After building R', we train the ranking model based on it.

Notably, the training set R' contains binary labels (i.e., 1 or 0), whereas the ranking models are expected to output continuous values, referred to as misclassification scores. To address this, we made certain modifications to the ranking algorithms we employed, such as the random forest. During the classification process, these algorithms calculate an intermediate value, which is used to decide whether an input belongs to a particular class. If the intermediate value exceeds a predefined threshold of 0.5 (which is configurable), the input is classified into the first class; otherwise, it is classified into the other class. Rather than outputting the binary label, we directly output the intermediate value, representing the misclassification score. This score indicates the likelihood of a test input being misclassified by the GNN model, with a higher score indicating a greater probability of misclassification.

4.4 Evaluation Design

To assess NodeRank, we enumerate various research questions (cf. Section 4.4.1), which explore the performance metric (cf. Section 4.4.2) for test inputs prioritzation on a diverse set of GNN subjects (cf. Section 4.4.4). Beyond the prioritization performance of NodeRank in uncovering model misclassification, we also consider the performance under adversarial settings (cf. Section 4.4.5). In this section, we also present how the design of the different variants of NodeRank(cf. Section 4.4.6), which vary based on the ensemble ranking strategy. Finally, information about implementation and configuration setup is provided in Section 4.4.7.

4.4.1 Research Questions

We investigate the following research questions:

- **RQ1: What is the effectiveness of NodeRank?**
  Building on studies in traditional software testing [57, 56], effective test prioritization techniques should be able to prioritize possibly-misclassified test inputs.
- **RQ2: How does NodeRank perform on adversarial inputs?**
  Graph adversarial attacks [139, 140] can induce GNN models to be confident in their however-incorrect predictions. Thus, existing confidence-based test prioritization approaches are likely to fail. We demonstrate the superior performance of NodeRank under such settings.
- **RQ3: How does NodeRank perform with different ensemble ranking strategies?**
  We investigate the performance of NodeRank variants implemented by considering three different ensemble learning techniques.
- **RQ4: Are all mutation feature categories useful in NodeRank?**
  We conduct an ablation study on NodeRank to assess the contribution of graph structure mutation features, node mutation features, and graph model mutation feature on the performance of NodeRank. Our ablation experiments follow prior work by Meyes *et al.* [151].
- **RQ5: Do the model mutation rules of NodeRank contribute to its effectiveness?**
  In the original NodeRank, for a given test input, we employ the killing approach from traditional mutation testing [152] to generate model mutation features. These features are then utilized to predict the misclassification probability for this input. However, the model mutation features generated by such a killing approach can contain information from both the model mutation rules and the randomness inherent in mutated model training, both of which can contribute to the effectiveness of NodeRank. In this research question, we aim to demonstrate that the model mutation rules actually contribute to the effectiveness of NodeRank by employing the killing approach in DeepCrime [60], which takes into account the training randomness of the mutated models during the killing process.
- **RQ6: How do the parameter ranges of the newly designed mutation operators impact the effectiveness of NodeRank?**
  In NodeRank, we developed a set of novel mutation operators tailored for GNNs. In this research question, we investigate how the parameter ranges of these newly designed mutation operators affect the performance of NodeRank.

4.4.2 Performance Metric

We evaluate the effectiveness of test prioritization based on the common Average Percentage of Fault-Detection (APFD) [11] metric. Specifically, higher APFD values indicate faster misclassification detection rates. Given a GNN model M under the test set T, the APFD values are calculated via Formula 4.6.

$$APFD = 1 - \frac{\sum_{i=1}^{k} o_i}{kn} + \frac{1}{2n} \tag{4.6}$$

where n is the number of test inputs in T; k is the number of test inputs in T that will be misclassified by M; o_i represents the position of the i_{th} misclassified test within the prioritized test set. When the sum of the index values for the first k

misclassified tests, i.e., $\sum_{i=1}^{k} o_i$, is small, it indicates that the prioritized test set has a higher order of the misclassified tests, leading to a larger APFD score. Consequently, a higher APFD score indicates better prioritization effectiveness.

Following prior work [6], we perform normalization on the APFD values, making them fall in the range of [0, 1] to facilitate comparison. We thus assume a test prioritization approach is better if its APFD value is closer to 1.

To conduct a more detailed evaluation, we employ the Percentage of Fault Detected (PFD) metric [6] to quantify the fault detection rate of each test prioritization approach across varying ratios of prioritized test inputs. High PFD values indicate higher effectiveness in identifying misclassified test inputs. PFD is calculated based on Formula 4.7.

$$PFD = \frac{F_c}{F_t} \tag{4.7}$$

where F_c is the number of misclassified test inputs that are correctly detected. F_t is the total number of misclassified test inputs.

In this study, we compare the PFD of NodeRank and the uncertainty-based test prioritization approaches against different ratios of prioritized tests. We use **PFD-n** to represent the first n% prioritized test inputs.

4.4.3 Compared Approaches

This study utilized five compared approaches, including a baseline approach (i.e., random selection) and four DNN test prioritization techniques. The selection of these methods was driven by several factors. Firstly, we aimed to consider approaches that could be feasibly adapted for GNN test prioritization. Secondly, the chosen techniques have been demonstrated as effective for DNNs in the existing literature [6, 9, 101]. Lastly, open-source implementations of these techniques are available.

- **DeepGini** [6] employs the Gini coefficient as a statistical measure of the likelihood of misclassification, thereby enabling the ranking of test inputs. The calculation of the Gini score is presented in Formula 4.8.

$$\xi(x) = 1 - \sum_{i=1}^{N} \left(p_i(x)\right)^2 \tag{4.8}$$

  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.
- **Vanilla Softmax** [9] calculates the difference between the value of 1 and the maximum activation probability in the output softmax layer. Formula 6.21 clearly depicts the calculation process.

$$\mathrm{V}(x) = 1 - \max_{c=1}^{C} l_c(x) \tag{4.9}$$

  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 [9] measures the difference between the predicted class and the second most confident class in softmax likelihood. PCS is calculated by Formula 4.10. Low PCS values indicate high probability of being misclassified.

$$P(x) = l_k(x) - l_j(x) \tag{4.10}$$

where $l_k(x)$ refers to the most confident prediction probability. $l_j(x)$ refers to the second most confident prediction probability.

- **Entropy** Entropy [9] measures uncertainty in a classification model's prediction for a given test by computing the entropy of the softmax likelihood.
- **GraphPrior** GraphPrior [108] is a test prioritization method specifically designed for GNNs. GraphPrior generates mutated models for GNNs and regards tests that kill many mutated models as more likely to be misclassified.
- **Random selection** [102] In random selection, the order of execution for test inputs is determined randomly.

4.4.4 GNN Subjects

4.4.4.1 Graph datasets

Our study utilizes four benchmark datasets commonly used in the field of graph neural networks (GNNs). The Cora and CiteSeer datasets are composed of machine learning publications, represented as nodes in a graph structure, with edges representing citation links between the publications. The PubMed dataset, on the other hand, contains biomedicine publications. The LastFM Asia Social Network dataset, consists of the relationships between users on the Last.fm music service in Asia, where users are represented as nodes and their mutual follower relationships are represented as edges. These datasets have been widely adopted in existing research on graph neural networks [153, 154, 155, 156, 157].

Overall, we built 124 subjects to evaluate the effectiveness of NodeRank, including 16 subjects of natural datasets and 108 subjects of adversarial datasets.

- **Cora** [15] Cora comprises 2,708 scientific publications and 5,429 links between them. Publications are considered nodes and are classified into seven classes.
- **CiteSeer** [15] CiteSeer is composed of 3,327 scientific publications and 4,732 links between them. Publications (nodes) are classified into six classes.
- **PubMed** [15] PubMed is composed of 19,717 diabetes-related publications and 44,338 links between them. Publications (nodes) are classified into three classes.
- **LastFM Asia Social Network** [158] LastFM Asia Social Network comprises 7,624 nodes and 27,806 edges.

4.4.4.2 GNN models

We consider four GNN models which have been widely studied in the literature of neural network testing, specifically under adversarial attacks.

- **Graph Convolutional Network (GCN)** [145] is a class of neural networks that use graph convolutions. GCN leverages the information of edges to aggregate node information to generate new node representations.
- **Graph Attention Network (GAT)** [146] introduces a graph attention layer to weigh the importance of different nodes within a neighborhood. Each node is assigned an attention score so that more important neighbors can be identified.
- **Topology Adaptive GCN (TAGCN)** [148] designs a set of fixed-size learnable filters to perform convolution operations on graphs. These filters adapt to the topology of the graph while it is scanned for convolution.
- **Graph Sample and Aggregate (GraphSAGE)** [147] generates node embeddings through sampling and aggregating features of neighbor nodes. For computational efficiency, GraphSAGE samples a fixed number of neighbors for each node.

4.4.5 Graph Adversarial Attacks

In RQ2, we aim to investigate the effectiveness of NodeRank on test inputs generated through diverse graph adversarial attacks. Graph adversarial attacks refer to the manipulation of the graph structure or node features to generate graph adversarial perturbations that fool the GNN models. To evaluate the performance of NodeRank against such attacks, we applied a range of adversarial attacks in our experiments. We introduced these attacks as follows.

- **Delete internally, connect externally (DICE)** [138] DICE randomly inserts or deletes an edge for each perturbation. DICE follows two crucial rules: 1) only removing edges between nodes that are from the same class, and 2) only inserting nodes that are from different classes.
- **Min-max attack (MMA)** [139] The min-max attack is a type of untargeted white-box GNN attack, which formulates the attack problem as a min-max optimization problem. In this setup, the inner maximization objective is to update the model's parameters (θ) by maximizing the attack loss, and it can be efficiently solved using gradient ascent. Meanwhile, the outer minimization is achieved using the Projected Gradient Descent (PGD) [159] algorithm, which iteratively perturbs the graph within a bounded ℓ_p norm constraint to ensure that the generated perturbations are not too large.
- **Node embedding attack-Add (NEAA)** [140] In the node embedding attack-add, attackers have the ability to manipulate the original graph structure by adding new edges while ensuring that a predetermined budget constraint is not exceeded.
- **Node embedding attack-Remove (NEAR)** [140] In the Node embedding attack-Remove, adversarial attacks are aimed at modifying the original graph structure by selectively removing edges while adhering to a budget constraint.
- **PGD attack (PGD)** [139] The PGD attack leverages the Projected Gradient Descent (PGD) algorithm to search for optimal structural perturbations to attack GNNs.
- **Random Attack-Add (RAA)** [141] RAA randomly adds edges to the input graph to generate perturbations.
- **Random Attack-Remove (RAR)** [141] RAR randomly removes edges to the input graph to generate perturbations.
- **Random Attack-Flip (RAF)** [141] RAF randomly flips edges to the input graph to generate perturbations.

4.4.6 Variants of NodeRank

In this paper, when using NodeRank, we refer to the approach that utilizes the Sum-based ensemble learning method (cf. Section 4.3.4) on top of the four considered base models, namely Logistic Regression [150], Random Forest [90], XGBoost [13], and LightGBM [89]. We also implemented two variants using the stacking-based, and voting-based ensemble methods.

4.4.6.1 NodeRank^S

With this variant, we implemented a stacking-based ensemble method, which uses meta-learning [160] to learn from the outputs of base ranking models to make more accurate predictions. Given a GNN model M that classified nodes into n classes and a test set T_{test}, NodeRank^S performs as follows: (1) first, each base ranking

model RM_i is trained using mutation features of the training input set T_{train} of M; (2) then, NodeRank uses the output of each ranking model to create a new dataset. More specifically, NodeRank^S inputs the mutation results of the training set to each ranking model to obtain the outputs. For each training input, NodeRank^S obtains four probability scores, which will be considered as new features, while the label is 1 or 0. Here, 1 means the training input is misclassified by the GNN model M, while 0 means the training input is correctly classified. Since the training set has ground truth for each input, in this way, we build a new dataset. (3) NodeRank^S uses the new dataset to train the meta-learner. Here, each input has four features, which are the outputs from the four ranking models. The ground truth is whether an input is misclassified by the GNN model M. (4) After training the meta-learner, NodeRank^S inputs the mutation results of the test set T_{test} to ranking models. Then, NodeRank^S inputs the outputs of ranking models to the meta-learner, which will provide a score for each test input in T_{test}. Based on the scores, NodeRank^S prioritizes all the test inputs.

4.4.6.2 NodeRank^V

With this variant, we implemented the majority voting-based ensemble learning method [161] to combine the prediction results of different ranking models. Majority voting sums the predictions for each class and returns the class with the majority vote as the ensemble prediction. Given a GNN model M and a test set T_{test}, NodeRank^V performs as follows: (1) first, each base ranking model RM_i is trained using mutation features of the training input set T_{train} of M; (2) For a test input in T_{test}, NodeRank^V inputs its mutation features to N ranking models, obtaining N scores (i.e., misclassification probabilities) for this input, denoted as $\{S_1, S_2, \cdots, S_N\}$. Then, NodeRank^S transforms each score into 0 or 1. Scores below 0.5 are converted to 0, otherwise to 1. In this way, NodeRank^S obtains an N-length vector for each input. For example, $\{0, 1, \cdots, 0\}$. NodeRank^S regards 1 voting for misclassification (i.e., the input will be misclassified by the GNN model M) and 0 voting for correct classification. (3) After voting, for each input, NodeRank^V sums its votes from all ranking models. NodeRank^V ranks all the test inputs based on their votes for misclassification.

4.4.7 Implementation and Configuration

We implemented NodeRank in Python based on the PyTorch [103] framework. We also integrate the available implementations of the compared approaches [6, 105, 88] into our experimental pipeline to adapt to the GNN prioritization problem. Regarding the GNN models selected as subjects in our study, the range of their accuracy is: GAT: 71%∼77%, GCN: 70%∼73%, GraphSAGE: 71%∼73%, TAGCN: 72%∼81%. Regarding our mutation rules, for the GNN model mutation, we generated 144 mutants on average. For graph structure mutation, we generated 265 mutants on average. For node feature mutation, we generated 147 mutants on average. Concerning the configurations of node mutation rules in the experiments of this paper, we made the following design choice: We slightly modify attributes, with an offset between 0.005 to 0.015.

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

GB RAM. Overall, our experiments involved 124 subjects, of which 16 subjects were based on natural inputs and 108 subjects were based on adversarial inputs.

4.5 Experimental Results

For each research question, we present the experimental objective, design, and results before discussing the findings.

Table 4.1: Effectiveness comparison among NodeRank, Random, DeepGini, VanillaSM, PCS, and Entropy in terms of the APFD values on natural datasets

Approach	CiteSeer				Cora				LastFM				PubMed			
	GAT	GCN	GraphSAGE	TAGCN	GAT	GCN	GraphSAGE	TAGCN	GAT	GCN	GraphSAGE	TAGCN	GAT	GCN	GraphSAGE	TAGCN
Random	0.4784	0.4893	0.4837	0.4882	0.4912	0.5268	0.4857	0.4782	0.4870	0.4981	0.5142	0.5049	0.4890	0.4972	0.5032	0.5123
DeepGini	0.6072	0.6197	0.6732	0.6260	0.7080	0.7037	0.7070	0.7650	0.5887	0.6991	0.7820	0.7368	0.6371	0.6995	0.6952	0.6140
VanillaSM	0.6519	0.6611	0.6831	0.6534	0.7325	0.7292	0.7283	0.7688	0.6696	0.7437	0.7850	0.7677	0.6630	0.7196	0.6981	0.6583
PCS	0.6528	0.6848	0.6767	0.6541	0.7132	0.7239	0.7303	0.7330	0.6925	0.7454	0.7463	0.7548	0.6569	0.6738	0.6660	0.6658
Entropy	0.6045	0.6181	0.6727	0.6165	0.7019	0.7007	0.7025	0.7564	0.5228	0.6411	0.7082	0.6011	0.6402	0.7004	0.6968	0.6155
GraphPrior	0.6754	0.6942	0.7103	0.6961	0.7853	0.7883	0.7651	0.7815	0.7746	0.7834	0.7914	0.7792	0.7546	0.7426	0.7534	0.7285
NodeRank	0.7319	0.7203	0.7325	0.7199	0.8326	0.8021	0.8121	0.8164	0.8146	0.8151	0.8063	0.8225	0.7714	0.7670	0.7895	0.7795

4.5.1 RQ1: Performance of NodeRank

Objective: We evaluate the performance of NodeRank in prioritizing test inputs for GNNs. To that end, we also compare NodeRank against five uncertainty-based test prioritization approaches.

Experimental design: We use our initial subjects (4 datasets and 4 GNN models, leading to 16 combinations of *natural inputs*, i.e., without any adversarial attacks introduced). Moreover, we compare NodeRank with 6 test prioritization approaches, which include 1 test prioritization method for GNNs (GraphPrior), 4 test prioritization methods for traditional DNNs (i.e., DeepGini, VanillaSM, PCS, and Entropy), and a baseline method (random selection). Specific details about these compared methods can be found in Section 4.4.3. All subjects are applied to NodeRank, as well as the six compared approaches. Beyond effectiveness, we also investigated the efficiency of NodeRank by analyzing the time cost of each step involved in its execution. Furthermore, due to the randomness in the GNN model training process, we conducted a statistical analysis to ensure the stability of our findings. Following the prior work [162], we repeated all the experiments 30 times. The following results are the averages obtained from the 30 repeated experiments.

To demonstrate the statistical significance of the improvement of NodeRank relative to the compared test prioritization approaches, we utilized the Mann-Whitney U test [163] to compute the p-value of the repeated experimental results. The Mann-Whitney U test is a statistical method used to determine whether there is a notable distinction between two sets of data distributions. The Mann-Whitney U test does not require the assumption of normal distribution for the data. Therefore, it can be used for both normal and non-normal distributed data. The Mann-Whitney U test transforms the data into ranks, calculates a test statistic based on these ranks, and uses this as a basis for computing the p-value to assess if there is a statistically significant difference between the two sets of data. A p-value < 0.05 is generally considered indicative of significance.

Furthermore, in addition to showcasing the average experimental results, we also evaluate the variability of these results in order to ensure a more fair comparison between the effectiveness of NodeRank and existing test prioritization approaches. The specific steps of these experiments are elucidated below:

- **Effectiveness distributions between NodeRank and the compared approaches** As previously mentioned, we conducted 30 repetitions of all experiments.

Subsequently, based on the results generated from these 30 repetitions, we used box plots to illustrate the distribution of results for various test prioritization methods. The rationale behind employing box plots is that: 1) they offer an intuitive representation of data distribution, including key statistics like the median, quartiles, and identification of outliers. This visual format enables a quick understanding of data characteristics; 2) Box plots offer a visual tool for easily comparing the distribution of experimental results across various test prioritization approaches. When multiple box plots are displayed side by side, the differences between them can be clearly exhibited.

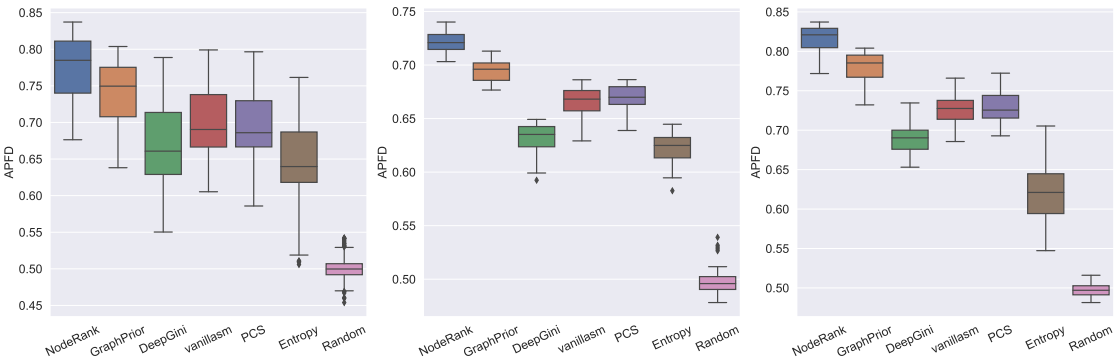
- **Confidence interval between NodeRank and the compared approaches** Based on the results of 30 repeated experiments, we calculated the confidence interval of each test prioritization approach. Following the existing study [164], we employed Formula 4.11 to compute the upper and lower bounds of the confidence interval. We calculated the confidence intervals for different test prioritization methods across two metrics (PFD and APFD) and two scenarios (natural and adversarial datasets).

$$\left(\bar{X} - Z_{\frac{\alpha}{2}}\frac{\sigma}{\sqrt{n}}, \bar{X} + Z_{\frac{\alpha}{2}}\frac{\sigma}{\sqrt{n}}\right) \tag{4.11}$$

where $\bar{X}$ represents the average value, σ represents the standard deviation, n represents the sample size, and $Z_{\alpha/2}$ represents the confidence coefficient.

Table 4.2: Performance improvement of NodeRank on the 16 initial subjects (i.e., 4 natural input sets on 4 GNN models)

Approach	# Best cases	Average APFD	Improvement(%)
Random	0	0.4954	58.11
DeepGini	0	0.6788	15.39
VanillaSM	0	0.7071	10.78
PCS	0	0.6981	12.20
Entropy	0	0.6513	20.27
GraphPrior	0	0.7502	4.41
NodeRank	16	0.7833	-



a) All natural subjects **b)** CiteSeer, TAGCN **c)** LastFM, GCN

Figure 4.2: Effectiveness distributions between NodeRank and the compared approaches on natural test inputs

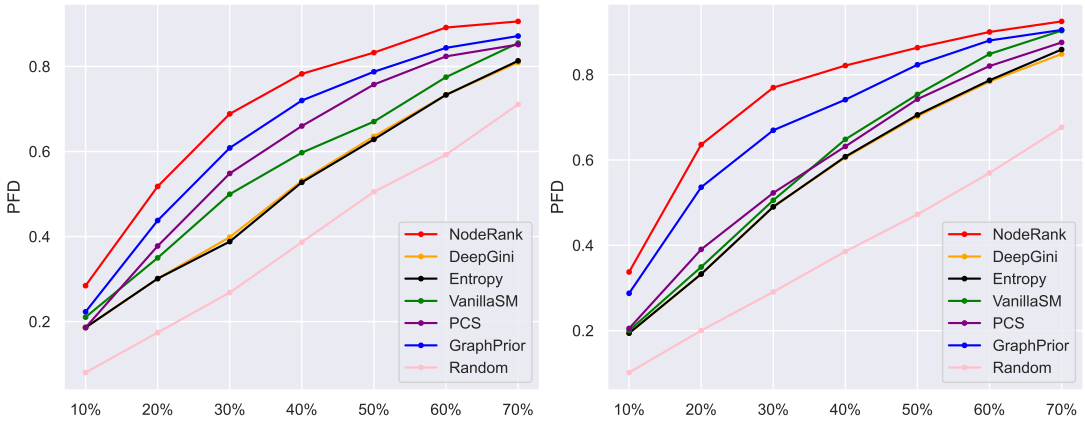
Results: The experimental results of RQ1 are presented in Table 4.1, Table 4.2, Table 4.3, Table 4.4, Table 4.5, Table 4.6, Table 4.7, Figure 4.2, and Figure 4.3. We highlight the approach with the highest effectiveness in grey to facilitate quick and easy interpretation of the results. Table 4.1 presents the APFD scores of NodeRank

Table 4.3: Average comparison results among NodeRank and the compared approaches in terms of PFD

Data	Approach	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60
CiteSeer	Random	0.0859	0.1758	0.2802	0.3859	0.4776	0.5753
	DeepGini	0.1999	0.3393	0.4609	0.5796	0.6849	0.7705
	VanillaSM	0.2104	0.3805	0.5254	0.6471	0.7253	0.8077
	PCS	0.2103	0.3858	0.5267	0.6478	0.7513	0.8217
	Entropy	0.1991	0.3402	0.4542	0.5772	0.6766	0.7683
	GraphPrior	0.2355	0.4431	0.6024	0.7253	0.7827	0.8235
	NodeRank	0.2684	0.5078	0.6790	0.7752	0.8267	0.8697
Cora	Random	0.0938	0.1955	0.2996	0.3893	0.4971	0.5822
	DeepGini	0.2555	0.4647	0.6193	0.7406	0.8293	0.8831
	VanillaSM	0.2718	0.4906	0.6461	0.7676	0.8541	0.9170
	PCS	0.2406	0.4386	0.6061	0.7513	0.8422	0.9103
	Entropy	0.2551	0.4627	0.6100	0.7325	0.8250	0.8759
	GraphPrior	0.3025	0.6021	0.7254	0.8013	0.8923	0.9215
	NodeRank	0.3434	0.6764	0.8682	0.9113	0.9342	0.9468
LastFM	Random	0.1021	0.1983	0.3041	0.4045	0.5039	0.5994
	DeepGini	0.2476	0.4549	0.6042	0.7128	0.7898	0.8548
	VanillaSM	0.2560	0.4939	0.6606	0.7814	0.8658	0.9177
	PCS	0.2253	0.4593	0.6527	0.7883	0.8698	0.9143
	Entropy	0.2472	0.4264	0.5190	0.6022	0.6705	0.7214
	GraphPrior	0.3015	0.5324	0.7612	0.8563	0.8746	0.9237
	NodeRank	0.3487	0.6833	0.8621	0.9083	0.9297	0.9473
PubMed	Random	0.1015	0.2023	0.3027	0.3989	0.4959	0.5957
	DeepGini	0.2344	0.4026	0.5463	0.6407	0.7226	0.7959
	VanillaSM	0.2270	0.4034	0.5649	0.6935	0.7851	0.8516
	PCS	0.1968	0.3811	0.5422	0.6640	0.7630	0.8313
	Entropy	0.2348	0.4028	0.5467	0.6424	0.7264	0.7994
	GraphPrior	0.3021	0.5163	0.6582	0.7535	0.8192	0.8746
	NodeRank	0.3463	0.6258	0.7744	0.8359	0.8748	0.9062

and the compared approaches on each subject (i.e., a combination of a natural dataset and GNN model). We see that NodeRank consistently outperforms all compared approaches on all 16 subjects (i.e., 16 Best cases for NodeRank). Moreover, the APFD range for NodeRank is 0.7199 to 0.8326, while GraphPrior (the test prioritization method specifically designed for GNNs) falls within the range of 0.6754 to 0.7883. Additionally, the APFD range for other test prioritization methods varies from 0.4784 to 0.7850. Table 4.2 presents an in-depth assessment of NodeRank's effectiveness in comparison to other approaches, including the number of best cases achieved by each approach, the average APFD, and the improvement that NodeRank offers over the compared methods. We see that the average APFD for NodeRank is 0.7883, while the average APFD for GraphPrior is 0.7502. In contrast, the average APFD range for other test prioritization methods falls between 0.4954 and 0.7071. When compared to GraphPrior, NodeRank exhibits an average improvement of 4.41%, while its improvement relative to other comparative methods ranges from 10.78% to 58.11%.

We present further evidence of the high effectiveness of NodeRank in the context of test prioritization by utilizing the PFD (Percentage of Fault Detected) metric. The corresponding experimental results are presented in Table 4.3. Our analysis demonstrates that NodeRank consistently surpasses GraphPrior, all the confidence-based approaches, and random selection in terms of average PFD, regardless of



a) CiteSeer, TAGCN **b)** PubMed, GAT

Figure 4.3: Test prioritization effectiveness among NodeRank and the compared approaches for CiteSeer with TAGCN and PubMed with GAT. X-Axis: the percentage of prioritized tests; Y-Axis: the percentage of detected misclassified tests.

Table 4.4: Confidence interval of NodeRank and the compared approaches in terms of APFD on natural test inputs

Approach	Lower Bound	Upper Bound
Random	0.4942	0.4966
DeepGini	0.6743	0.6832
VanillaSM	0.7032	0.7109
PCS	0.6945	0.7016
Entropy	0.6467	0.6558
GraphPrior	0.7467	0.7536
NodeRank	0.7798	0.7867

the proportion of prioritized tests. Furthermore, the effectiveness of NodeRank is visually apparent in Figure 4.3. In the figure, NodeRank is represented by the red line, GraphPrior by the blue line, and the baseline method by the pink line. It is evident that NodeRank consistently outperforms GraphPrior, all the confidence-based approaches, and the baseline. These experimental findings further confirm the high effectiveness of NodeRank.

Table 4.6 presents the results of statistical analysis. We use the Mann–Whitney U test [163] as the metric to calculate the p-value of the experimental results. Our objective is to demonstrate that the improvement of NodeRank over other testing methods is statistically significant. Within Table 4.6, we see that the range of p-values is from 7.7245×10^{-7} to 0.0092. These values are all less than 0.05, indicating that the improvement of NodeRank compared to other test prioritization methods is statistically significant.

Moreover, Figure 4.2 presents and compares the effectiveness (in terms of APFD) of NodeRank with other test prioritization methods using box plots. The box plots highlight the distribution of results of multiple repeated experiments for NodeRank and other test prioritization methods. In Figure 4.2, we see that, in terms of the median, the median APFD value of NodeRank exceeds that of other test prioritization methods across all natural datasets. Moreover, in the presented two specific examples shown in the box plots, which respectively correspond to subject CiteSeer, TAGCN, and subject LastFM, GCN, we can also see that the median of NodeRank from repeated experiments is the highest.

Table 4.5: Confidence interval of NodeRank and the compared approaches in terms of PFD on natural test inputs

Approach	PFD-10		PFD-20		PFD-30		PFD-40		PFD-50		PFD-60	
	Lower	Upper	Lower	Upper	Lower	Upper	Lower	Upper	Lower	Upper	Lower	Upper
Random	0.0917	0.1008	0.1887	0.1989	0.2907	0.3024	0.3879	0.4007	0.4872	0.4994	0.5817	0.5933
DeepGini	0.2280	0.2411	0.4101	0.4197	0.5523	0.5628	0.6620	0.6743	0.7496	0.7609	0.8197	0.8305
VanillaSM	0.2346	0.2482	0.4373	0.4477	0.5928	0.6032	0.7175	0.7265	0.8016	0.8129	0.8675	0.8791
PCS	0.2124	0.2231	0.4112	0.4222	0.5760	0.5877	0.7071	0.7183	0.7998	0.8111	0.8637	0.8737
Entropy	0.2285	0.2398	0.4012	0.4128	0.5282	0.5378	0.6319	0.6426	0.7186	0.7309	0.7870	0.7953
GraphPrior	0.2811	0.2920	0.5175	0.5291	0.6826	0.6912	0.7773	0.7899	0.8359	0.8477	0.8814	0.8923
NodeRank	0.3215	0.3322	0.6167	0.6294	0.7910	0.8014	0.8534	0.8631	0.8865	0.8967	0.9118	0.9218

Table 4.6: Statistical analysis on natural test inputs (in terms of p-value under the Mann–Whitney U test)

Mann–Whitney U test	NodeRrank vs Random	NodeRank vs DeepGini	NodeRank vs VanillaSM	NodeRank vs PCS	NodeRank vs Entropy	NodeRank vs GraphPrior
p-value	7.7245×10^{-7}	1.1175×10^{-5}	9.5154×10^{-5}	2.5438×10^{-5}	1.6231×10^{-6}	0.0092

Table 4.7: Time cost of NodeRank and the compared approaches

Time cost	Approach						
	NodeRank	Random	GraphPrior	DeepGini	VanillaSM	PCS	Entropy
Mutant generation	35 min	-	35 min	-	-	-	-
Feature extraction	30 s	-	20 s	-	-	-	-
Ranking model training	3 min	-	3 min	-	-	-	-
Prediction	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s

Regarding the quartile range, NodeRank's quartile range (i.e., the height of the box) exhibits some variations across different datasets, but overall, its upper quartile is higher than that of other methods. Analyzing outliers, the box plots do not show significant outliers, indicating that NodeRank's performance across different datasets is relatively stable, with no extreme cases of inefficiency. In summary, we conclude that NodeRank outperforms all compared testing prioritization methods in terms of APFD based on the distribution of data from multiple experimental results. This demonstrates that NodeRank exhibits better effectiveness in test prioritization compared to other methods.

Furthermore, we calculated the confidence intervals for all test prioritization methods, and the experimental results are presented in Table 4.4 and Table 4.5. In Table 4.4, we see that NodeRank's APFD has the highest lower and upper bounds compared to other test prioritization methods, with values of 0.7798 and 0.7867, respectively. Notably, NodeRank's lower bound (0.7798) even exceeds the upper bounds of all other comparative methods. GraphPrior's upper bound is 0.7536, while the upper bounds for other test prioritization methods range from 0.4966 to 0.7109. Table 4.5 exhibits the confidence intervals of all test prioritization methods in terms of PFD. The gray highlights indicate the test prioritization approaches that achieve the maximum PFD in this scenario. In Table 4.5, we see that NodeRank also demonstrates the highest lower and upper bounds in terms of PFD compared to other test prioritization methods when prioritizing different ratios of tests. These experimental findings highlight that, in terms of confidence intervals, NodeRank's effectiveness exceeds that of the comparative test prioritization methods.

In addition to its effectiveness, we also present an analysis of NodeRank's efficiency in Table 4.7. We offer a comprehensive breakdown of the time taken by each step in NodeRank and compare it with GraphPrior, the confidence-based test prioritization methods, and the baseline approach (random selection). As shown in Table 4.7, the time required for NodeRank is divided into four parts: mutant generation, feature

extraction, ranking model training, and NodeRank prediction. Among these steps, mutant generation is found to be the most time-consuming, taking approximately 35 minutes, followed by ranking model training, which takes approximately 3 minutes. Overall, NodeRank requires a total time of approximately 38 minutes. However, it is worth noting that the prediction time of NodeRank is extremely fast, taking less than 1s once the ranking model is trained, and the mutation features are extracted. The overall runtime of GraphPrior is similar to NodeRank, approximately 38 minutes. In contrast, the confidence-based test prioritization methods have an overall runtime of less than 1 second. While NodeRank is less efficient than the uncertainty-based test prioritization approaches (which takes less than 1s), its time cost remains acceptable compared to the prohibitively expensive manual labeling.

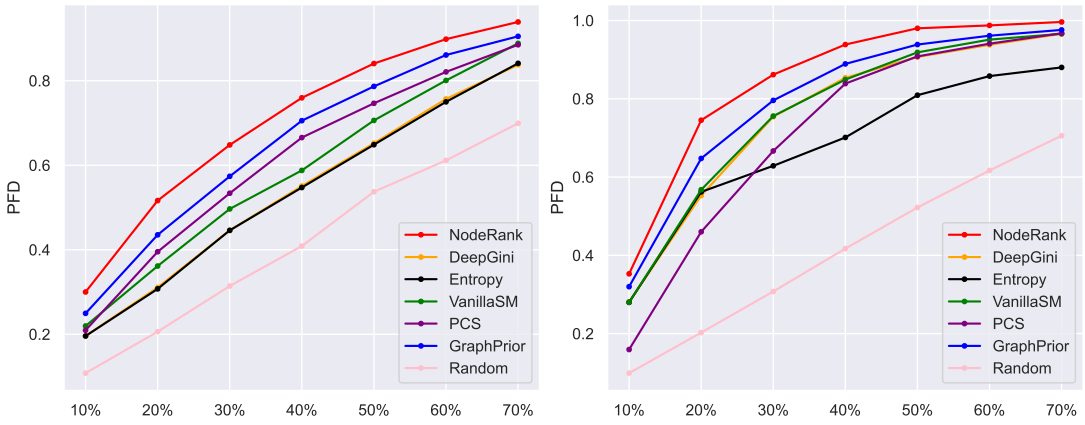
Answer to RQ1: *On natural test inputs, NodeRank consistently exhibits better effectiveness compared to GraphPrior, all confidence-based approaches, and the baseline method across all subjects, as evident from both APFD and PFD metrics.*

In terms of APFD, NodeRank showcases an average improvement of 4.41% and 58.11% over the compared approaches. Additionally, the efficiency of NodeRank is within an acceptable range, thereby demonstrating its practical usefulness.

4.5.2 RQ2: Prioritization of Adversarial Inputs

Table 4.8: Test prioritization performance (APFD scores) on DICE-based graph adversarial test inputs

Approach	CiteSeer				Cora				LastFM				PubMed			
	GAT	GCN	GraphSAGE	TAGCN	GAT	GCN	GraphSAGE	TAGCN	GAT	GCN	GraphSAGE	TAGCN	GAT	GCN	GraphSAGE	TAGCN
Random	0.4867	0.4986	0.4937	0.5153	0.4938	0.5250	0.4817	0.5100	0.5031	0.4907	0.5004	0.4973	0.4997	0.4953	0.4929	0.4940
DeepGini	0.5893	0.6059	0.6550	0.6168	0.6878	0.6873	0.7061	0.7269	0.5883	0.6897	0.7685	0.7136	0.6363	0.6726	0.6846	0.6145
VanillaSM	0.6159	0.6418	0.6695	0.6335	0.7058	0.7113	0.7189	0.7359	0.6559	0.7294	0.7720	0.7502	0.6567	0.6875	0.6875	0.6488
PCS	0.6058	0.6494	0.6639	0.6304	0.6803	0.6974	0.7065	0.7066	0.6696	0.7289	0.7342	0.7428	0.6453	0.6379	0.6538	0.6509
Entropy	0.5874	0.6044	0.6536	0.6123	0.6827	0.6839	0.7025	0.7175	0.5364	0.6478	0.6961	0.5828	0.6384	0.6731	0.6860	0.6157
GraphPrior	0.7013	0.6955	0.7143	0.6745	0.7525	0.7436	0.7515	0.7537	0.7652	0.7561	0.7827	0.7732	0.7067	0.7037	0.7051	0.7038
NodeRank	0.7249	0.7128	0.7402	0.7171	0.8054	0.7963	0.8031	0.7859	0.8034	0.8059	0.8054	0.8040	0.7543	0.7465	0.7752	0.7699



a) MMA, CiteSeer, GCN **b)** PGD, LastFM, GraphSAGE

Figure 4.4: Test prioritization effectiveness among NodeRank and the compared approaches for CiteSeer with GCN attacked by MMA and LastFM with GraphSAGE attacked by PGD. X-Axis: the percentage of prioritized tests; Y-Axis: the percentage of detected miscalssified tests.

Objective: We evaluate the effectiveness of NodeRank on adversarial test inputs. We assume that natural test inputs (cf. RQ1) can easily discriminate which ones are more likely to reveal bugs. In contrast, with adversarial inputs, by construction, they are all generated to make the probability of the wrong classification label as high as

Table 4.9: Overall comparison results on graph adversarial datasets

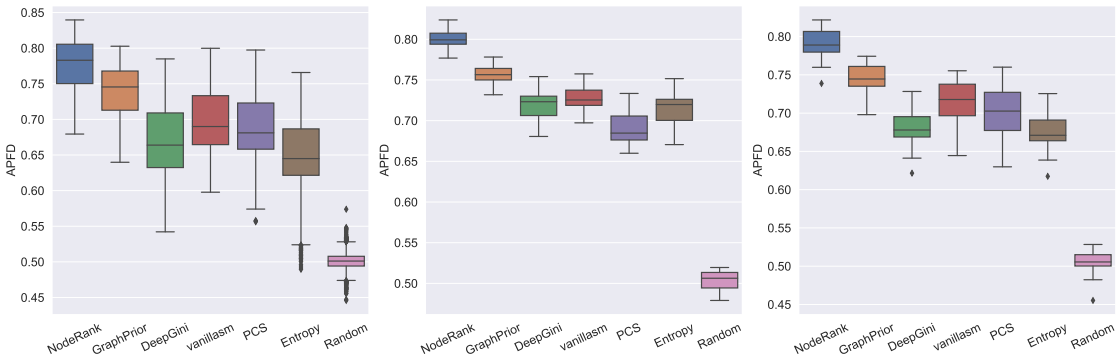
Attack	Approach	Average performance score (APFD)				Improvement(of APFD) of NodeRank over the compared approaches			
		GAT	GCN	GraphSAGE	TAGCN	GAT	GCN	GraphSAGE	TAGCN
DICE	Random	0.4958	0.5024	0.4922	0.5042	55.71%	52.35%	58.68%	52.56%
	DeepGini	0.6254	0.6639	0.7036	0.6679	23.44%	15.29%	11.03%	15.17%
	VanillaSM	0.6586	0.6925	0.7120	0.6921	17.22%	10.53%	9.69%	11.14%
	PCS	0.6503	0.6784	0.6896	0.6827	18.71%	12.82%	13.25%	12.67%
	Entropy	0.6112	0.6523	0.6846	0.6321	26.31%	17.34%	14.08%	21.69%
	GraphPrior	0.7314	0.7247	0.7384	0.7263	5.54%	5.61%	5.76%	5.91%
	NodeRank	0.7720	0.7654	0.7810	0.7692	-	-	-	-
MMA	Random	0.5283	0.5128	0.5161	0.4775	46.60%	47.62%	50.34%	60.27%
	DeepGini	0.6591	0.6616	0.6960	0.6950	17.51%	14.42%	11.48%	10.12%
	VanillaSM	0.6857	0.6923	0.7093	0.7104	12.95%	9.35%	9.39%	7.73%
	PCS	0.6662	0.6980	0.7034	0.6952	16.26%	8.45%	10.31%	10.08%
	Entropy	0.6551	0.6591	0.6938	0.6881	18.23%	14.85%	11.83%	11.22%
	GraphPrior	0.7258	0.7126	0.7392	0.7266	6.70%	6.23%	4.96%	5.32%
	NodeRank	0.7745	0.7570	0.7759	0.7653	-	-	-	-
NEAA	Random	0.5079	0.5053	0.5009	0.4938	54.54%	54.98%	59.19%	60.33%
	DeepGini	0.6311	0.6869	0.7238	0.6864	24.37%	14.00%	10.17%	15.34%
	VanillaSM	0.6686	0.7170	0.7301	0.7172	17.39%	9.22%	9.23%	10.39%
	PCS	0.6630	0.6931	0.7062	0.7032	18.39%	12.99%	12.95%	12.59%
	Entropy	0.6114	0.6708	0.7005	0.6410	28.38%	16.74%	13.83%	23.51%
	GraphPrior	0.7421	0.7368	0.7521	0.7433	5.76%	6.28%	6.02%	6.51%
	NodeRank	0.7849	0.7831	0.7974	0.7917	-	-	-	-
NEAR	Random	0.4936	0.5016	0.4946	0.5134	61.93%	58.19%	62.15%	56.62%
	DeepGini	0.6426	0.6955	0.7241	0.7005	24.39%	14.09%	10.76%	14.79%
	VanillaSM	0.6850	0.7237	0.7337	0.7269	16.69%	9.64%	9.31%	10.62%
	PCS	0.6841	0.7061	0.7096	0.7144	16.84%	12.38%	13.02%	12.56%
	Entropy	0.6206	0.6771	0.6976	0.6523	28.79%	17.19%	14.97%	23.27%
	GraphPrior	0.7352	0.7548	0.7625	0.7581	8.71%	5.12%	5.18%	6.06%
	NodeRank	0.7993	0.7935	0.8020	0.8041	-	-	-	-
PGD	Random	0.5043	0.5114	0.5026	0.5090	56.81%	52.07%	57.02%	53.14%
	DeepGini	0.6378	0.6764	0.7239	0.7137	23.99%	14.98%	9.02%	9.22%
	VanillaSM	0.6839	0.7133	0.7336	0.7304	15.63%	9.03%	7.58%	6.72%
	PCS	0.6805	0.7187	0.7176	0.7133	16.21%	8.21%	9.98%	9.28%
	Entropy	0.6169	0.6593	0.6997	0.6644	28.19%	17.96%	12.79%	17.32%
	GraphPrior	0.7491	0.7324	0.7403	0.7387	5.56%	6.19%	6.60%	5.52%
	NodeRank	0.7908	0.7778	0.7892	0.7795	-	-	-	-
RAA	Random	0.4981	0.4951	0.5027	0.5018	53.88%	55.10%	55.06%	54.15%
	DeepGini	0.6299	0.6660	0.7084	0.6709	21.69%	15.30%	10.04%	15.29%
	VanillaSM	0.6596	0.6964	0.7160	0.6972	16.21%	10.27%	8.87%	10.94%
	PCS	0.6459	0.6824	0.6931	0.6836	18.67%	12.53%	12.47%	13.15%
	Entropy	0.6166	0.6553	0.6910	0.6360	24.31%	17.18%	12.81%	21.62%
	GraphPrior	0.7046	0.7261	0.7312	0.7258	8.78%	5.75%	6.60%	6.57%
	NodeRank	0.7665	0.7679	0.7795	0.7735	-	-	-	-
RAF	Random	0.4990	0.4964	0.5003	0.5004	54.31%	55.00%	56.69%	54.66%
	DeepGini	0.6199	0.6660	0.7074	0.6724	24.21%	15.53%	10.81%	15.10%
	VanillaSM	0.6519	0.6971	0.7157	0.6984	18.12%	10.37%	9.53%	10.81%
	PCS	0.6415	0.6829	0.6937	0.6828	20.03%	12.67%	13.00%	13.34%
	Entropy	0.6062	0.6550	0.6882	0.6374	27.02%	17.47%	13.91%	21.42%
	GraphPrior	0.7109	0.7257	0.7369	0.7281	8.34%	6.02%	6.37%	6.29%
	NodeRank	0.7702	0.7694	0.7839	0.7739	-	-	-	-
RAR	Random	0.5134	0.5015	0.5043	0.5024	52.84%	53.84%	56.61%	56.35%
	DeepGini	0.6312	0.6765	0.7063	0.6895	24.32%	14.04%	11.82%	13.92%
	VanillaSM	0.6723	0.7080	0.7153	0.7115	16.72%	8.97%	10.42%	10.40%
	PCS	0.6700	0.6990	0.6994	0.7039	17.12%	10.37%	12.93%	11.59%
	Entropy	0.6144	0.6620	0.6875	0.6542	27.72%	16.54%	14.88%	20.07%
	GraphPrior	0.7403	0.7325	0.7421	0.7362	5.99%	5.32%	6.42%	6.69%
	NodeRank	0.7847	0.7715	0.7898	0.7855	-	-	-	-

possible. Thus, a test input prioritization on adversarial inputs may be challenged in ranking them adequately. Yet, such prioritization is still necessary to ensure a fast assessment of GNN model robustness.

Experimental design: To investigate the effectiveness of NodeRank on adversarial datasets, we generated adversarial test inputs using eight graph adversarial attack

Table 4.10: Confidence interval of NodeRank and the compared approaches in terms of APFD on DICE-based graph adversarial test inputs

Approach	Lower Bound	Upper Bound
Random	0.4976	0.4996
DeepGini	0.6610	0.6693
VanillaSM	0.6848	0.6925
PCS	0.6715	0.6788
Entropy	0.6409	0.6491
GraphPrior	0.7271	0.7334
NodeRank	0.7686	0.7749



a) All adversarial subjects **b)** RAR, Cora, GAT **c)** NEAA, Cora, GCN

Figure 4.5: Effectiveness distributions between NodeRank and the compared approaches on adversarial test inputs

methods [138, 141, 140]. We set the attack level to 0.3, which indicates that 30% of the test inputs in the test set are adversarial tests. It is worth noting that a high attack level, such as 90%, would result in a significant proportion of adversarial test inputs. Under such circumstances, any prioritization method could potentially select a larger number of bug cases, making it difficult to effectively demonstrate the efficacy of NodeRank. Thus, to ensure a proper evaluation of NodeRank and the compared approaches, we selected a reasonable attack level (i.e., 0.3), which effectively limits the proportion of adversarial test inputs.

Eventually, we construct 108 subjects (i.e., a combination of a GNN model and an adversarial inputs set). Consistent with the experimental design employed in RQ1, we evaluate the prioritization effectiveness of NodeRank and the compared approaches using both the APFD and PFD metrics. Similar to RQ1, we conducted 30 repetitions of all experiments and reported the average outcomes. Aside from presenting the average experimental findings, we assessed the variability of these results to ensure a fairer comparison between the effectiveness of NodeRank and existing test prioritization methods. Detailed steps for these experiments can be found in the experimental design of RQ1 (refer to Section 4.5.1).

Results: The experimental results of RQ2 are presented in Table 4.8, Table 4.9, Table 4.10, Table 4.11, Table 4.12, Table 4.13, Table 4.14, Figure 4.4, and Figure 4.5. Table 4.8 presents the APFD scores of NodeRank and the compared approaches on DICE-based graph adversarial inputs. Again, NodeRank performs the best across all subjects. Experiment results on all the subjects are available on our GitHub[2]

Table 4.9 presents the average APFD values for NodeRank and the compared

[2]https://github.com/yinghuali/NodeRank/tree/main/results

Table 4.11: Average comparison results among NodeRank and the compared approaches on adversarial data in terms of PFD

Attack	Approach	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60
DICE	Random	0.0977	0.1943	0.2930	0.3959	0.4985	0.5994
	DeepGini	0.2119	0.3904	0.5359	0.6451	0.7359	0.8173
	VanillaSM	0.2167	0.4054	0.5601	0.6894	0.7865	0.8582
	PCS	0.1945	0.3708	0.5359	0.6704	0.7750	0.8524
	Entropy	0.2113	0.3835	0.5149	0.6191	0.7064	0.7843
	GraphPrior	0.2587	0.5023	0.6856	0.7835	0.8344	0.8923
	NodeRank	0.2876	0.5594	0.7569	0.8473	0.8968	0.9267
MMA	Random	0.1059	0.2126	0.3165	0.4071	0.5129	0.6036
	DeepGini	0.2261	0.4012	0.5465	0.6621	0.7572	0.8353
	VanillaSM	0.2379	0.4246	0.5860	0.6964	0.7885	0.8628
	PCS	0.2153	0.3976	0.5589	0.6900	0.7891	0.8649
	Entropy	0.2248	0.3987	0.5424	0.6567	0.7496	0.8284
	GraphPrior	0.2788	0.5081	0.6782	0.7823	0.8438	0.8935
	NodeRank	0.2972	0.5710	0.7391	0.8277	0.8801	0.9155
NEAA	Random	0.0974	0.2011	0.3017	0.3980	0.5045	0.6033
	DeepGini	0.2306	0.4167	0.5634	0.6724	0.7646	0.8359
	VanillaSM	0.2327	0.4302	0.5939	0.7215	0.8167	0.8857
	PCS	0.2022	0.3949	0.5652	0.6996	0.7977	0.8725
	Entropy	0.2307	0.4090	0.5342	0.6393	0.7276	0.7947
	GraphPrior	0.2845	0.5127	0.6943	0.7857	0.8571	0.9146
	NodeRank	0.3055	0.5936	0.7978	0.8811	0.9184	0.9409
NEAR	Random	0.0960	0.2001	0.2965	0.4005	0.5014	0.6057
	DeepGini	0.2435	0.4366	0.5813	0.6934	0.7724	0.8416
	VanillaSM	0.2482	0.4556	0.6148	0.7408	0.8262	0.8905
	PCS	0.2143	0.4163	0.5933	0.7219	0.8206	0.8819
	Entropy	0.2423	0.4274	0.5501	0.6551	0.7317	0.7954
	GraphPrior	0.3071	0.5788	0.7834	0.8123	0.8662	0.9014
	NodeRank	0.3435	0.6520	0.8266	0.8796	0.9084	0.9337
PGD	Random	0.0988	0.2033	0.2992	0.4067	0.5123	0.6161
	DeepGini	0.2350	0.4247	0.5702	0.6826	0.7706	0.8429
	VanillaSM	0.2487	0.4558	0.6106	0.7315	0.8162	0.8824
	PCS	0.2249	0.4209	0.5878	0.7259	0.8200	0.8845
	Entropy	0.2341	0.4120	0.5393	0.6432	0.7333	0.7982
	GraphPrior	0.2835	0.5241	0.6836	0.7826	0.8543	0.8992
	NodeRank	0.3171	0.6083	0.7795	0.8531	0.8996	0.9304
RAA	Random	0.1035	0.1991	0.2959	0.3985	0.4956	0.6013
	DeepGini	0.2110	0.3917	0.5392	0.6491	0.7483	0.8273
	VanillaSM	0.2166	0.4052	0.5686	0.6990	0.7933	0.8627
	PCS	0.1929	0.3734	0.5347	0.6755	0.7787	0.8552
	Entropy	0.2108	0.3862	0.5192	0.6253	0.7203	0.7960
	GraphPrior	0.2545	0.4523	0.6834	0.7436	0.8521	0.9034
	NodeRank	0.2826	0.5518	0.7587	0.8543	0.9011	0.9271
RAF	Random	0.1007	0.1997	0.2970	0.3954	0.4968	0.5925
	DeepGini	0.2108	0.3915	0.5335	0.6487	0.7412	0.8188
	VanillaSM	0.2158	0.4044	0.5669	0.6972	0.7921	0.8606
	PCS	0.1918	0.3728	0.5343	0.6705	0.7754	0.8516
	Entropy	0.2105	0.3852	0.5136	0.6228	0.7142	0.7873
	GraphPrior	0.2465	0.5014	0.6547	0.7362	0.8357	0.9033
	NodeRank	0.2816	0.5535	0.7611	0.8628	0.9048	0.9326
RAR	Random	0.0970	0.2033	0.3058	0.4053	0.5117	0.6062
	DeepGini	0.2273	0.4100	0.5573	0.6614	0.7532	0.8226
	VanillaSM	0.2400	0.4305	0.5854	0.7114	0.8022	0.8650
	PCS	0.2124	0.4057	0.5691	0.6981	0.8003	0.8677
	Entropy	0.2268	0.4044	0.5376	0.6347	0.7227	0.7909
	GraphPrior	0.2833	0.5344	0.6836	0.7843	0.8561	0.9013
	NodeRank	0.3222	0.6164	0.7853	0.8533	0.8951	0.9210

approaches, as well as the average improvement of NodeRank over the compared approaches across different adversarial attacks. We can see that, across all cases, NodeRank consistently outperforms GraphPrior, confidence-based approaches, and random selection. Specifically, NodeRank achieves an average APFD ranging from

Table 4.12: Confidence interval of NodeRank and the compared approaches in terms of PFD on adversarial datasets

Attack	Approach	PFD-10		PFD-20		PFD-30		PFD-40		PFD-50		PFD-60	
		Lower	Upper	Lower	Upper	Lower	Upper	Lower	Upper	Lower	Upper	Lower	Upper
DICE	Random	0.0951	0.1006	0.1915	0.1972	0.2903	0.2974	0.3912	0.3984	0.4948	0.5026	0.5949	0.6028
	DeepGini	0.2090	0.2160	0.3857	0.3952	0.5309	0.5392	0.6420	0.6495	0.7331	0.7397	0.8150	0.8205
	VanillaSM	0.2130	0.2206	0.4015	0.4090	0.5553	0.5637	0.6871	0.6922	0.7836	0.7904	0.8551	0.8626
	PCS	0.1911	0.1972	0.3675	0.3745	0.5331	0.5407	0.6667	0.6753	0.7708	0.7770	0.8475	0.8559
	Entropy	0.2067	0.2151	0.3801	0.3872	0.5114	0.5180	0.6161	0.6235	0.7033	0.7112	0.7814	0.7871
	GraphPrior	0.2542	0.2622	0.4995	0.5046	0.6835	0.6902	0.7794	0.7874	0.8303	0.8375	0.8898	0.8944
	NodeRank	0.2835	0.2908	0.5571	0.5618	0.7519	0.7618	0.8445	0.8511	0.8929	0.9012	0.9218	0.9306
MMA	Random	0.1019	0.1103	0.2084	0.2167	0.3115	0.3206	0.4049	0.4101	0.5094	0.5155	0.6014	0.6073
	DeepGini	0.2214	0.2285	0.3972	0.4056	0.5426	0.5510	0.6574	0.6652	0.7549	0.7599	0.8306	0.8383
	VanillaSM	0.2342	0.2412	0.4216	0.4279	0.5817	0.5888	0.6921	0.6997	0.7838	0.7908	0.8594	0.8676
	PCS	0.2130	0.2192	0.3940	0.4009	0.5558	0.5617	0.6856	0.6938	0.7860	0.7937	0.8602	0.8693
	Entropy	0.2222	0.2271	0.3951	0.4022	0.5390	0.5446	0.6537	0.6606	0.7449	0.7521	0.8246	0.8325
	GraphPrior	0.2755	0.2818	0.5054	0.5128	0.6760	0.6810	0.7790	0.7854	0.8388	0.8474	0.8889	0.8982
	NodeRank	0.2946	0.3019	0.5679	0.5752	0.7360	0.7434	0.8252	0.8304	0.8776	0.8831	0.9107	0.9193
NEAA	Random	0.0926	0.1010	0.1969	0.2051	0.2991	0.3038	0.3948	0.4005	0.4995	0.5073	0.5983	0.6069
	DeepGini	0.2261	0.2332	0.4130	0.4215	0.5605	0.5670	0.6703	0.6759	0.7610	0.7666	0.8323	0.8388
	VanillaSM	0.2286	0.2365	0.4252	0.4325	0.5891	0.5969	0.7184	0.7248	0.8142	0.8213	0.8834	0.8887
	PCS	0.1996	0.2058	0.3906	0.3991	0.5629	0.5696	0.6964	0.7029	0.7935	0.8006	0.8692	0.8761
	Entropy	0.2282	0.2336	0.4045	0.4134	0.5319	0.5391	0.6357	0.6415	0.7243	0.7324	0.7903	0.7996
	GraphPrior	0.2812	0.2866	0.5099	0.5165	0.6900	0.6968	0.7826	0.7885	0.8541	0.8615	0.9111	0.9189
	NodeRank	0.3018	0.3090	0.5915	0.5957	0.7930	0.8005	0.8771	0.8846	0.9140	0.9213	0.9362	0.9437
NEAR	Random	0.0932	0.0985	0.1956	0.2033	0.2932	0.3008	0.3980	0.4035	0.4980	0.5063	0.6022	0.6097
	DeepGini	0.2394	0.2463	0.4318	0.4408	0.5779	0.5833	0.6897	0.6963	0.7679	0.7752	0.8380	0.8437
	VanillaSM	0.2444	0.2530	0.4509	0.4585	0.6116	0.6179	0.7381	0.7442	0.8240	0.8300	0.8859	0.8947
	PCS	0.2105	0.2171	0.4134	0.4196	0.5908	0.5978	0.7198	0.7241	0.8180	0.8249	0.8769	0.8856
	Entropy	0.2389	0.2449	0.4244	0.4301	0.5468	0.5549	0.6526	0.6573	0.7288	0.7343	0.7910	0.7998
	GraphPrior	0.3042	0.3098	0.5740	0.5827	0.7797	0.7870	0.8093	0.8146	0.8614	0.8685	0.8979	0.9056
	NodeRank	0.3387	0.3467	0.6480	0.6548	0.8220	0.8295	0.8754	0.8842	0.9045	0.9114	0.9303	0.9362
PGD	Random	0.0956	0.1023	0.1984	0.2056	0.2963	0.3036	0.4032	0.4114	0.5088	0.5169	0.6124	0.6208
	DeepGini	0.2301	0.2387	0.4197	0.4276	0.5672	0.5734	0.6800	0.6867	0.7668	0.7740	0.8388	0.8472
	VanillaSM	0.2463	0.2531	0.4517	0.4605	0.6065	0.6148	0.7276	0.7339	0.8114	0.8187	0.8798	0.8854
	PCS	0.2222	0.2290	0.4186	0.4230	0.5851	0.5914	0.7238	0.7284	0.8161	0.8235	0.8812	0.8884
	Entropy	0.2306	0.2382	0.4092	0.4166	0.5347	0.5435	0.6400	0.6456	0.7287	0.7360	0.7953	0.8029
	GraphPrior	0.2794	0.2868	0.5193	0.5282	0.6793	0.6866	0.7793	0.7869	0.8500	0.8576	0.8967	0.9019
	NodeRank	0.3129	0.3205	0.6053	0.6128	0.7770	0.7842	0.8507	0.8574	0.8974	0.9023	0.9282	0.9351
RAA	Random	0.1004	0.1079	0.1949	0.2012	0.2916	0.2979	0.3960	0.4022	0.4935	0.4996	0.5981	0.6053
	DeepGini	0.2083	0.2141	0.3871	0.3944	0.5367	0.5421	0.6457	0.6533	0.7452	0.7510	0.8234	0.8318
	VanillaSM	0.2140	0.2188	0.4027	0.4081	0.5637	0.5727	0.6949	0.7032	0.7908	0.7959	0.8598	0.8670
	PCS	0.1894	0.1959	0.3710	0.3765	0.5310	0.5396	0.6706	0.6798	0.7748	0.7832	0.8515	0.8588
	Entropy	0.2080	0.2143	0.3830	0.3893	0.5161	0.5227	0.6208	0.6273	0.7164	0.7240	0.7930	0.8008
	GraphPrior	0.2496	0.2588	0.4491	0.4566	0.6793	0.6862	0.7403	0.7462	0.8482	0.8541	0.8991	0.9074
	NodeRank	0.2776	0.2875	0.5492	0.5564	0.7543	0.7626	0.8515	0.8578	0.8980	0.9036	0.9229	0.9306
RAF	Random	0.0978	0.1043	0.1971	0.2021	0.2937	0.2995	0.3913	0.3995	0.4947	0.5007	0.5898	0.5951
	DeepGini	0.2087	0.2149	0.3882	0.3946	0.5311	0.5374	0.6457	0.6517	0.7369	0.7457	0.8155	0.8224
	VanillaSM	0.2118	0.2197	0.4001	0.4070	0.5619	0.5691	0.6936	0.7011	0.7894	0.7944	0.8558	0.8643
	PCS	0.1875	0.1947	0.3703	0.3774	0.5307	0.5389	0.6657	0.6739	0.7732	0.7800	0.8479	0.8558
	Entropy	0.2062	0.2151	0.3808	0.3890	0.5104	0.5164	0.6205	0.6249	0.7101	0.7169	0.7843	0.7896
	GraphPrior	0.2442	0.2491	0.4973	0.5040	0.6523	0.6570	0.7319	0.7395	0.8309	0.8394	0.9008	0.9062
	NodeRank	0.2793	0.2842	0.5491	0.5558	0.7564	0.7654	0.8606	0.8648	0.9026	0.9076	0.9295	0.9369
RAR	Random	0.0940	0.1015	0.1992	0.2063	0.3030	0.3102	0.4031	0.4095	0.5072	0.5141	0.6023	0.6109
	DeepGini	0.2229	0.2310	0.4059	0.4137	0.5532	0.5607	0.6576	0.6642	0.7484	0.7553	0.8176	0.8249
	VanillaSM	0.2377	0.2449	0.4256	0.4329	0.5816	0.5902	0.7074	0.7160	0.7977	0.8044	0.8615	0.8678
	PCS	0.2099	0.2160	0.4008	0.4083	0.5656	0.5716	0.6951	0.7004	0.7953	0.8049	0.8649	0.8723
	Entropy	0.2236	0.2298	0.4001	0.4071	0.5350	0.5405	0.6326	0.6378	0.7182	0.7271	0.7866	0.7929
	GraphPrior	0.2784	0.2861	0.5320	0.5376	0.6815	0.6870	0.7809	0.7866	0.8527	0.8606	0.8983	0.9038
	NodeRank	0.3201	0.3254	0.6128	0.6212	0.7831	0.7875	0.8510	0.8556	0.8904	0.8977	0.9183	0.9242

0.7570 to 0.8041, whereas GraphPrior averages between 0.7046 and 0.7625. The remaining testing prioritization methods show APFD ranges from 0.4922 to 0.7337. In terms of improvement over GraphPrior, NodeRank demonstrates an average improvement ranging from 4.69% to 8.78%. NodeRank's improvement over the other testing prioritization methods varies from 6.72% to 62.15%.

Table 4.13 presents the results of statistical analysis on adversarial datasets. The adopted approach (Mann-Whitney U test method) for calculating the p-value is explained in RQ1 (Section 4.5.1). Within Table 4.13, we see that the range of p-values is from 2.4541×10^{-8} to 0.0057. All these values fall below 0.05, indicating that the improvement of NodeRank in comparison to other test prioritization methods is statistically significant.

Table 4.10 presents the confidence intervals for all test prioritization methods in relation to the metric APFD. We see that NodeRank's APFD has the highest lower and upper bounds compared to other test prioritization methods. Specifically, the lower bound is 0.7686, and the upper bound is 0.7749. These experimental results

Table 4.13: Statistical analysis on adversarial datasets (in terms of p-value under the Mann–Whitney U test)

Attack	NodeRrank vs Random	NodeRank vs DeepGini	NodeRank vs VanillaSM	NodeRank vs PCS	NodeRank vs Entropy	NodeRank vs GraphPrior
DICE	3.3978×10^{-8}	6.8803×10^{-7}	1.2979×10^{-5}	2.9367×10^{-6}	1.7078×10^{-7}	0.0021
MMA	2.4541×10^{-8}	6.4702×10^{-5}	4.6045×10^{-4}	1.2353×10^{-4}	2.0828×10^{-5}	0.0014
NEAA	4.5746×10^{-8}	3.3978×10^{-7}	2.3978×10^{-5}	4.2542×10^{-6}	1.4537×10^{-7}	0.0002
NEAR	2.6732×10^{-8}	6.1731×10^{-7}	1.1088×10^{-6}	5.3228×10^{-7}	3.3978×10^{-7}	0.0009
PGD	3.3274×10^{-8}	2.6274×10^{-5}	4.6045×10^{-4}	5.1867×10^{-5}	1.3448×10^{-6}	0.0017
RAA	8.2331×10^{-8}	3.4582×10^{-7}	5.5223×10^{-6}	1.7497×10^{-6}	1.1088×10^{-7}	0.0016
RAF	3.4582×10^{-8}	1.0307×10^{-6}	1.4624×10^{-5}	2.2701×10^{-6}	1.7078×10^{-7}	0.0057
RAR	6.4691×10^{-8}	3.9739×10^{-7}	5.5223×10^{-6}	1.3448×10^{-6}	9.5885×10^{-7}	0.0041

Table 4.14: Confidence interval of NodeRank and the compared approaches in terms of APFD on adversarial test inputs

Approach	Lower Bound	Upper Bound
Random	0.5013	0.5022
DeepGini	0.6705	0.6738
VanillaSM	0.6958	0.6987
PCS	0.6832	0.6861
Entropy	0.6501	0.6534
GraphPrior	0.7319	0.7344
NodeRank	0.7760	0.7783

underscore that in terms of APFD and considering confidence intervals, NodeRank demonstrates better effectiveness compared to other test prioritization methods.

In addition to the APFD metric, we also computed the PFD of NodeRank and compared approaches under adversarial attack scenarios, and the results are presented in Table 4.11 and Figure 4.4. As shown in Table 4.11, NodeRank outperformed the compared approaches regarding PFD values for all attacks and any prioritization ratio of test inputs. Notably, NodeRank detected more than 90% of the bugs when approximately 50% of the test inputs were prioritized.

Furthermore, Figure 4.4 offers two visual examples for assessing the effectiveness of NodeRank compared to other approaches on the CiteSeer and LastFM datasets. In the figure, NodeRank is represented by a red line, GraphPrior by a blue line, and the baseline method by a pink line. We see that NodeRank consistently outperforms GraphPrior, as well as all confidence-based approaches and the baseline method. These experimental results demonstrate that the effectiveness of NodeRank exceeds that of all compared approaches under adversarial attack scenarios, indicating its efficacy in detecting bugs in adversarial datasets.

The box plot in Figure 4.5 illustrates NodeRank's effectiveness (in terms of APFD) compared to other test prioritization methods using box plots on adversarial datasets. It presents the distribution of results from multiple repeated experiments for both NodeRank and the compared approaches. In Figure 4.5, we see that, across all adversarial datasets, NodeRank's median effectiveness, as indicated by the median line within the box, surpasses that of other methods.

Regarding the quartile range, NodeRank's quartile range (i.e., the height of the box) exhibits some variations across different datasets, but overall, its upper quartile is higher than that of other methods. This difference is particularly noticeable in the subjects "RAR, Cora, GAT" and "NEAA, Cora, GCN". In terms of the outliers, we see that the box plots do not show significant outliers, indicating that

NodeRank's performance across different datasets is relatively stable. Based on the above experimental results, we conclude that NodeRank outperforms all compared testing prioritization methods in terms of APFD based on the distribution of data from multiple experimental results. This demonstrates that NodeRank exhibits higher effectiveness in test prioritization compared to other methods on adversarial datasets.

Moreover, Table 4.12 and Table 4.14 displays the confidence intervals of all test prioritization methods. Table 4.12 displays the confidence intervals of all test prioritization methods in terms of PFD. We see that, in terms of PFD, NodeRank also demonstrates the highest lower and upper bounds compared to other test prioritization approaches when prioritizing different ratios of tests. These experimental findings emphasize that, from the perspective of confidence intervals, NodeRank shows higher effectiveness compared to other test prioritization methods.

Table 4.14 displays the confidence intervals in terms of APFD. In Table 4.14, NodeRank's APFD shows the highest lower and upper bounds compared to other test prioritization methods, with values of 0.7760 and 0.7783, respectively. Remarkably, NodeRank's lower bound (0.7760) even surpasses the upper bounds of all other comparative methods. GraphPrior's upper bound is 0.7344, while the upper bounds for other test prioritization methods range from 0.5022 to 0.6987.

Answer to RQ2: *On adversarial test inputs, NodeRank consistently demonstrates better effectiveness in comparison to GraphPrior, all confidence-based approaches, and the baseline method across all subjects in terms of both the APFD and PFD metrics. Regarding APFD, NodeRank exhibits an average improvement of 4.96% and 62.15% over the compared methods.*

4.5.3 RQ3: Influence of Ensemble Learning Methods

Objective. We investigate the impact of ensemble learning strategies on NodeRank's effectiveness in the learning-to-rank process.

Experimental design. We employ NodeRank and its variants, namely NodeRank^V and NodeRank^S (cf. Section 4.4.6 for details), to prioritize test inputs for both natural and adversarial scenarios, and evaluate their effectiveness in terms of APFD. These variants differ in the ensemble learning strategies used in the learning-to-rank process.

Results. Table 4.15 presents the average effectiveness of NodeRank and its variants, along with several compared approaches, on both natural and adversarial datasets. The upper part shows the average effectiveness under different models, while the bottom part shows the average effectiveness across different datasets. From Table 4.15, we can observe that the average effectiveness of NodeRank and its variants outperform all the compared approaches (i.e., GraphPrior, the confidence-based approaches and random selection) in each case. Additionally, the effectiveness of NodeRank is comparatively better than their variants. Across different GNN models, NodeRank performs the best in 100% of the cases on natural data. Furthermore, on the adversarial data, NodeRank also outperforms in 100% of the cases. From the perspective of datasets, on natural data, NodeRank performs better than all the variants in each case. On adversarial data, NodeRank has the highest average effectiveness across all adversarial datasets. Overall, the final average effectiveness of NodeRank is 0.7833 and 0.7772 on natural and adversarial datasets, respectively. These experi-

Table 4.15: Performance (APFD scores) of NodeRank variants associated to different ensemble learning strategies
(#BC ⇔ #Best cases) and (Avg ⇔ Average APFD score)

Approach	Natural inputs					Adversarial inputs				
	#BC	GAT	GCN	GraphSAGE	TAGCN	#BC	GAT	GCN	GraphSAGE	TAGCN
Random	0	0.4864	0.5028	0.4967	0.4959	0	0.5037	0.5023	0.5009	0.5014
DeepGini	0	0.6353	0.6805	0.7144	0.6855	0	0.6325	0.6737	0.7115	0.6850
VanillaSM	0	0.6792	0.7134	0.7236	0.7120	0	0.6686	0.7045	0.7202	0.7089
PCS	0	0.6789	0.7069	0.7048	0.7020	0	0.6610	0.6934	0.7003	0.6961
Entropy	0	0.6174	0.6650	0.6950	0.6474	0	0.6167	0.6607	0.6921	0.6477
GraphPrior	0	0.7475	0.7421	0.7501	0.7463	0	0.7298	0.7307	0.7428	0.7354
NodeRank^V	0	0.7607	0.7505	0.7505	0.7481	0	0.7537	0.7483	0.7558	0.7475
NodeRank^S	0	0.7551	0.7495	0.7512	0.7561	0	0.7516	0.7447	0.7527	0.7533
NodeRank	16	0.7876	0.7761	0.7851	0.7846	108	0.7795	0.7731	0.7872	0.7802

Approach	with Natural inputs					with Adversarial inputs				
	CiteSeer	Cora	LastFM	Pubmed	Avg	CiteSeer	Cora	LastFM	Pubmed	Avg
Random	0.4849	0.4954	0.5011	0.5004	0.4955	0.5028	0.5064	0.4990	0.4991	0.5018
DeepGini	0.6315	0.7209	0.7017	0.6615	0.6788	0.6277	0.7082	0.6927	0.6604	0.6722
VanillaSM	0.6623	0.7397	0.7415	0.6847	0.7071	0.6524	0.7244	0.7318	0.6807	0.6973
PCS	0.6671	0.7251	0.7348	0.6656	0.6981	0.6511	0.7043	0.7257	0.6578	0.6847
Entropy	0.6279	0.7154	0.6183	0.6632	0.6562	0.6253	0.7027	0.6173	0.6618	0.6518
GraphPrior	0.6842	0.7801	0.7821	0.7448	0.7478	0.6735	0.7624	0.7637	0.7332	0.7332
NodeRank^V	0.6845	0.7873	0.7909	0.7471	0.7525	0.6941	0.7745	0.7839	0.7397	0.7481
NodeRank^S	0.6917	0.7845	0.7893	0.7465	0.7532	0.6881	0.7742	0.7864	0.7398	0.7471
NodeRank	0.7261	0.8158	0.8146	0.7768	0.7833	0.7286	0.8011	0.8093	0.7690	0.7772

Table 4.16: Feature ablation study results

Approach	with Natural inputs					with Adversarial inputs				
	CiteSeer	Cora	LastFM	Pubmed	**Avg**	CiteSeer	Cora	LastFM	Pubmed	**Avg**
Random prioritization (b/c No features)	0.4849	0.4954	0.5011	0.5004	**0.4955**	0.5028	0.5064	0.4990	0.4991	**0.5018**
DeepGini	0.6315	0.7209	0.7017	0.6615	**0.6788**	0.6277	0.7082	0.6927	0.6604	**0.6722**
VanillaSM	0.6623	0.7397	0.7415	0.6847	**0.7071**	0.6524	0.7244	0.7318	0.6807	**0.6973**
PCS	0.6671	0.7251	0.7348	0.6656	**0.6981**	0.6511	0.7043	0.7257	0.6578	**0.6847**
Entropy	0.6279	0.7154	0.6183	0.6632	**0.6562**	0.6253	0.7027	0.6173	0.6618	**0.6518**
GraphPrior	0.6842	0.7801	0.7821	0.7448	**0.7478**	0.6735	0.7624	0.7637	0.7332	**0.7332**
NodeRank$_{NFM}$	0.5828	0.6328	0.5933	0.6085	**0.6044**	0.5710	0.6200	0.5897	0.6030	**0.5959**
NodeRank$_{NFM+GSM}$	0.6315	0.6796	0.7025	0.6452	**0.6647**	0.6569	0.6876	0.7106	0.6504	**0.6764**
NodeRank (i.e., NodeRank$_{NFM+GSM+GMM}$)	0.7261	0.8158	0.8146	0.7768	**0.7833**	0.7286	0.8011	0.8093	0.7690	**0.7772**

mental results demonstrate that the sum-based ensemble learning strategies used in NodeRank is more suitable for test prioritization.

Answer to RQ3: *On both natural and adversarial datasets, NodeRank offers a better effectiveness, in terms of APFD, over other variants. We also note that any variant of NodeRank outperforms all the compared approaches in GNN test prioritization.*

4.5.4 RQ4: Ablation Study of Mutation Operators

Objective: We investigate the effect of each category of mutation operators (i.e., GSM, NFM, and GMM). To this end, we analyze the contributions of the features generated by each type of mutation operator and conduct corresponding ablation studies. We proceed as proposed by Meyes et al. [151]: We measure the impact of a component on an ML system by removing or replacing this component and observing whether the performance of the ML system is affected. The objective is not to comprehensively check which feature set combinations provide good performance but rather to check that each set contributes to the performance.

Experimental design: We assume that the node mutation features (NFM) as a key

component of the NodeRank approach. Then, the graph structure mutation (GSM) features, which are obtained from the dataset, are considered the next most important feature set. Finally, the graph model mutation (GMM) features are considered as the first that can be removed in the ablation study, following the process in [151]. The experimental steps for checking the contributions of each subset of mutation features to the performance of NodeRank are thus as follows:

1. We compute the test prioritization performance of NodeRank when all mutation features are used.

2. We compute the test prioritization performance of a variant of NodeRank where the ranking model is learned with vectors that do not consider GMM features.

3. We compute the test prioritization performance of a variant of NodeRank where the ranking model is learned only with NFM feature vectors (i.e., by removing the GMM and GSM).

4. Finally, we also consider the case where no features are used. NodeRank, therefore, does not implement ensemble learning to rank. Instead, we consider a random ranking approach to prioritize the test set.

Note that we do not attempt to perform experiments that compare the value of the different feature sets. Indeed, the mutation space of GNNs is complex, and mutations of different types can produce feature vectors of various sizes, which may implicitly impact the learning performance, making any performance comparison biased or uninformative.

Results: The results of the ablation experiment are reported in Table 4.16. As expected, the Random prioritization approach, which employs no mutation features for learning to rank, performs the worst in terms of APFD. In contrast, the NodeRank approach that learns to rank by incorporating all three mutation rule sets (pertaining to nodes, graph structure, and graph model) exhibits the highest performance. Remarkably, the exclusion of graph model mutation features leads to a decline in learning performance by approximately 17.84% and 14.90% in terms of APFD on natural and adversarial datasets, respectively. On the other hand, employing only node mutation features yields a significant improvement over Random prioritization, with a performance gain of approximately 21.98% and 18.75% in terms of APFD on natural and adversarial datasets, respectively.

Moreover, by comparing against the performance of uncertainty-based DNN test prioritization approaches and GraphPrior, we note that the combinations of the three categories of mutation features were necessary to achieve state-of-the-art performance in GNN test prioritization.

Answer to RQ4: *The design choice in NodeRank to include all three types of mutation operators was effective. Indeed, although the node mutation operator can enable NodeRank to outperform random prioritization, it is the combination of NFM, GSM, and GMM operators that together lead to the SOTA performance of NodeRank.*

Table 4.17: Effectiveness (APFD scores) of NodeRank's variants. ($\text{NodeRank}_{withoutGMM}$ does not generate mutated models. NodeRank_{Random} does not use model mutation rules to generate mutated models. $\text{NodeRank}_{DeepCime}$ uses model mutation rules to generate mutated models)

Approach	Natural inputs					Adversarial inputs				
	#BC	GAT	GCN	GraphSAGE	TAGCN	#BC	GAT	GCN	GraphSAGE	TAGCN
$\text{NodeRank}_{withoutGMM}$	0	0.6607	0.6467	0.6701	0.6812	0	0.6772	0.6634	0.6826	0.6965
NodeRank_{Random}	0	0.6892	0.7143	0.7185	0.7162	0	0.6847	0.7112	0.7152	0.7144
$\text{NodeRank}_{DeepCrime}(\text{effectSize} \geq 0.3)$	0	0.7465	0.7448	0.7571	0.7597	0	0.7440	0.7416	0.7575	0.7542
$\text{NodeRank}_{DeepCrime}(\text{effectSize} \geq 0.4)$	0	0.7472	0.7445	0.7573	0.7599	14	0.7439	0.7417	0.7569	0.7543
$\text{NodeRank}_{DeepCrime}(\text{effectSize} \geq 0.5)$	8	0.7480	0.7449	0.7558	0.7602	7	0.7439	0.7418	0.7571	0.7542
$\text{NodeRank}_{DeepCrime}(\text{effectSize} \geq 0.6)$	0	0.7472	0.7444	0.7568	0.7601	38	0.7441	0.7420	0.7572	0.7541
$\text{NodeRank}_{DeepCrime}(\text{effectSize} \geq 0.7)$	2	0.7444	0.7402	0.7605	0.7603	15	0.7425	0.7398	0.7576	0.7540
$\text{NodeRank}_{DeepCrime}(\text{effectSize} \geq 0.8)$	2	0.7414	0.7353	0.7571	0.7622	21	0.7412	0.7372	0.7557	0.7540
$\text{NodeRank}_{DeepCrime}(\text{effectSize} \geq 0.9)$	4	0.7383	0.7331	0.7573	0.7601	13	0.7398	0.7338	0.7533	0.7537

Approach	with Natural inputs					with Adversarial inputs				
	CiteSeer	Cora	LastFM	Pubmed	Avg	CiteSeer	Cora	LastFM	Pubmed	Avg
$\text{NodeRank}_{withoutGMM}$	0.6315	0.6796	0.7025	0.6452	0.6647	0.6569	0.6876	0.7106	0.6504	0.6764
NodeRank_{Random}	0.6686	0.7382	0.7452	0.6859	0.7095	0.6619	0.7460	0.7498	0.6676	0.7063
$\text{NodeRank}_{DeepCrime}(\text{effectSize} \geq 0.3)$	0.6995	0.7785	0.7734	0.7565	0.7520	0.7012	0.7677	0.7694	0.7495	0.7470
$\text{NodeRank}_{DeepCrime}(\text{effectSize} \geq 0.4)$	0.6998	0.7787	0.7736	0.7568	0.7521	0.7017	0.7667	0.7694	0.7495	0.7469
$\text{NodeRank}_{DeepCrime}(\text{effectSize} \geq 0.5)$	0.6989	0.7794	0.7739	0.7567	0.7522	0.7012	0.7670	0.7695	0.7496	0.7468
$\text{NodeRank}_{DeepCrime}(\text{effectSize} \geq 0.6)$	0.6993	0.7787	0.7737	0.7568	0.7521	0.7015	0.7672	0.7696	0.7497	0.7471
$\text{NodeRank}_{DeepCrime}(\text{effectSize} \geq 0.7)$	0.6991	0.7797	0.7706	0.7560	0.7513	0.7016	0.7669	0.7682	0.7478	0.7461
$\text{NodeRank}_{DeepCrime}(\text{effectSize} \geq 0.8)$	0.6967	0.7753	0.7692	0.7547	0.7490	0.7005	0.7664	0.7659	0.7457	0.7446
$\text{NodeRank}_{DeepCrime}(\text{effectSize} \geq 0.9)$	0.6965	0.7746	0.7661	0.7515	0.7472	0.6991	0.7646	0.7640	0.7432	0.7427

4.5.5 RQ5: Investigating the Contributions of Model Mutation Rules on NodeRank Effectiveness

Objective: In this research question, our aim is to demonstrate that the model mutation rules of NodeRank actually contribute to its effectiveness. In the original NodeRank, we utilize the killing approaches in traditional mutation analysis for DNNs. This killing process is used to generate model mutation features of a given test input. The features are then utilized to predict the misclassification probability of this input. In this process, the model mutation features generated by killing may contain information resulting from model mutation rules and randomness in model training, both of which may contribute to the effectiveness of NodeRank. In this research question, by utilizing the killing approach in DeepCrime [60], which considers the training randomness in the process of the killing, we aim to demonstrate that the model mutation rules actually contribute to the effectiveness of NodeRank.

Experimental design: To demonstrate the aforementioned objective, we designed three types of variants of NodeRank: 1) $\text{NodeRank}_{\text{DeepCrime}}$, a variant that utilizes DeepCrime's killing method to mitigate the influence of randomness when generating model mutation features. DeepCrime's killing approach takes into account the training randomness of the mutated model. Specifically, this killing approach requires repeating the training process n times for both the original model $N = \langle N_1, \ldots, N_n \rangle$ and its mutated model $M = \langle M_1, \ldots, M_n \rangle$. A test is considered killed if the difference between the outputs of the original and mutated models is statistically significant with non-negligible and non-small effect size. 2) $\text{NodeRank}_{\text{Random}}$, which does not incorporate model mutation rules and solely relies on random generation of model mutation features, and 3) $\text{NodeRank}_{\text{withoutGMM}}$, which does not utilize model mutation features. We validated whether model mutation rules contribute to the effectiveness of NodeRank by comparing the effectiveness of these three variants. If $\text{NodeRank}_{\text{DeepCrime}}$ outperformed both $\text{NodeRank}_{\text{Random}}$ and $\text{NodeRank}_{\text{withoutGMM}}$, we consider that the model mutation rules contribute to the effectiveness of NodeRank.

In the subsequent sections, we first describe the detailed implementation of DeepCrime. Then, we present the details of the variants of NodeRank and how we leverage these variants to demonstrate that model mutation rules contribute to NodeRank's effectiveness.

1) Implementation of DeepCrime

Given an original GNN model N and a test t, the DeepCrime approach follows the following method to determine whether a test is "killed".

❶ For the original GNN model N, we repeated its training process n times, resulting in n GNN models: $\langle N_1, \ldots, N_n \rangle$. Similarly, for the mutated model M, we repeated its training process n times, obtaining $\langle M_1, \ldots, M_n \rangle$. Consistent with previous research [60], we set $n = 20$ in our experiments.

❷ For $\langle N_1, \ldots, N_n \rangle$, we used each GNN model to make predictions on the test t and obtained the predicted classifications for t from each model. Similarly, for $\langle M_1, \ldots, M_n \rangle$, we used each mutated model to predict the test t and obtained the predicted classifications for t from each model.

❸ Prior work [60] suggests that the mutated model M is considered "killed" if, for the given test t, the difference between the output of the original and mutated models, denoted as $A_N(t) = \langle A_{N_1}, \ldots, A_{N_n} \rangle$ and $A_M(t) = \langle A_{M_1}, \ldots, A_{M_n} \rangle$, is statistically significant with a non-negligible and non-small effect size. Therefore, we measure whether the mutated model M is "killed" using Formula 4.12.

$$isKill(N, M, t) = \begin{cases} \text{true} & \text{if effectSize}(A_N(\text{t}), A_M(\text{t})) \geq \beta \\ & \text{and } p\_\text{value}(A_N(\text{t}), A_M(\text{t})) < \alpha \\ \text{false} & \text{otherwise} \end{cases} \tag{4.12}$$

In Formula 4.12, $isKilled$ indicates whether the test t "kills" the mutated model M. $A_N(t)$ represents a series of predictions (outputs) for test t from the set of models $\langle N_1, \ldots, N_n \rangle$. Similarly, $A_M(t)$ refers to $\langle A_{M_1}, \ldots, A_{M_n} \rangle$, representing the set of predictions (outputs) for test t from the set of models $\langle M_1, \ldots, M_n \rangle$.

The term "effect size" [106] quantitatively measures the difference between two distributions of APFD results. One commonly used measure of effect size is Cohen's d. This value can be interpreted using thresholds provided by Cohen [165]: $|d| < 0.2$ indicates a "negligible" effect, $|d| < 0.5$ indicates a "small" effect, $|d| < 0.8$ indicates a "medium" effect, and otherwise, it is considered a "large" effect. The prior study on DNN mutation analysis [60] pointed out that the effect size should be non-small. The β can have an impact on the effectiveness of the killing method DeepCrime in the context of NodeRank for test prioritization. In our experiments, we set β to be 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, and 0.9, covering a wide range of effect sizes. By adopting different effect size values, we can observe whether, under different effect sizes, the effectiveness of the NodeRank variants using DeepCrime and mutation rules is better than the variant not using mutation rules but instead randomly generating mutated models, thereby better validating the contributions of model mutation rules.

2) Variants of NodeRank

In the following, we explain how we design variants of NodeRank and how we utilize them to demonstrate the effectiveness of NodeRank's model mutation rule.

❶ In the first step, we generate three types of variants of NodeRank: $\text{NodeRank}_{\text{withoutGMM}}$, $\text{NodeRank}_{\text{Random}}$, and $\text{NodeRank}_{\text{DeepCrime}}$. Regarding $\text{NodeRank}_{\text{DeepCrime}}$, we set different values for the effect size, ranging from 0.3 to 0.9. This aims to measure the effectiveness of $\text{NodeRank}_{\text{DeepCrime}}$ across varying effect sizes, providing a

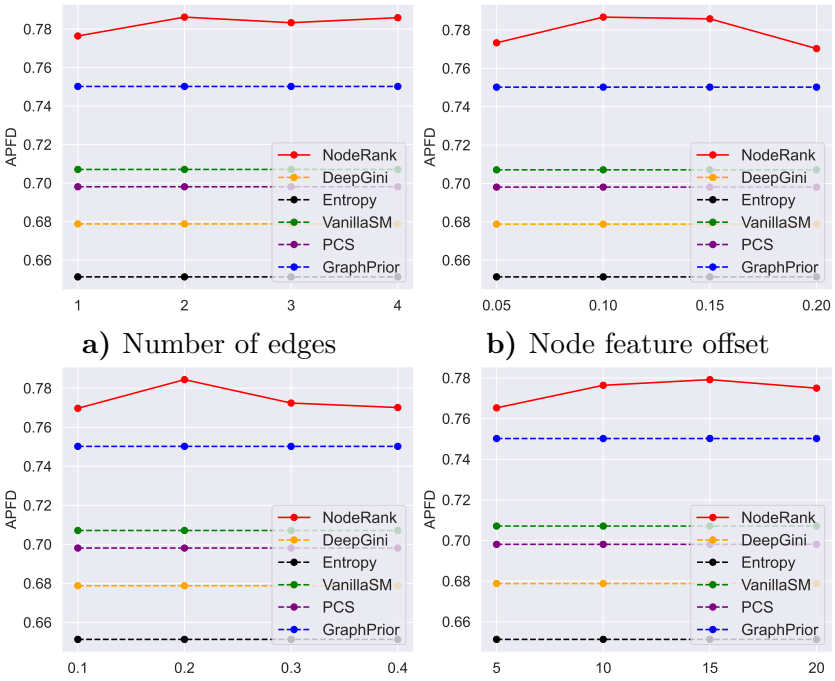
clearer demonstration of the effectiveness of our model mutation rules.

❷ $\text{NodeRank}_{\text{withoutGMM}}$ does not utilize model mutation features for test prioritization. All other workflow processes in this variant remain consistent with the original NodeRank.

❸ $\text{NodeRank}_{\text{Random}}$ utilizes model mutation features for test prioritization. However, the generated mutated models do not correspond to actual mutations; instead, it chooses to obtain different but equivalent GNN models as mutated models. Due to the randomness in training GNN models (such as the random initialization of model weights before training), different initializations can lead to different optimization paths during training, resulting in different weights at the end of training. Therefore, under the same configuration and operating conditions, the generated models can vary. In $\text{NodeRank}_{\text{Random}}$, we generated a series of equivalent GNN models as mutated models using the same configuration and operating conditions. All other workflow processes in this variant remain consistent with the original NodeRank.

❹ $\text{NodeRank}_{\text{DeepCrime}}$ uses DeepCrime's mutation killing approach as the killing approach to generate model mutation features for test prioritization. All other workflow processes in this variant remain consistent with the original NodeRank.

In summary, $\text{NodeRank}_{\text{withoutGMM}}$ represents NodeRank without the use of model mutation features. $\text{NodeRank}_{\text{Random}}$ incorporates model mutation features, but the mutated models are not generated by model mutation rules; instead, they alter the initial random seed to produce different but equivalent GNN models as the mutated models. $\text{NodeRank}_{\text{DeepCrime}}$ utilizes model mutation rules to generate mutated models. After completing the above process, we consider that if the effectiveness of $\text{NodeRank}_{\text{DeepCrime}}$ is higher than that of $\text{NodeRank}_{\text{Random}}$ and $\text{NodeRank}_{\text{withoutGMM}}$, the model mutation rules operated in $\text{NodeRank}_{\text{DeepCrime}}$ contribute to its effectiveness.



a) Number of edges **b)** Node feature offset

c) Negative slope **d)** Hidden channel

Figure 4.6: Impact of mutation operator parameters in NodeRank

Results: Table 4.17 presents the experimental results for RQ5. We highlighted

the approach with the highest effectiveness in grey to facilitate quick and easy interpretation of the results. The table above showcases the average effectiveness of $\text{NodeRank}_{\text{withoutGMM}}$, $\text{NodeRank}_{\text{Random}}$, and $\text{NodeRank}_{\text{DeepCrime}}$ across different subjects in terms of models. The table below shows the average effectiveness in terms of datasets.

The two tables show that in each case, $\text{NodeRank}_{\text{DeepCrime}}$ performs the best. Furthermore, each variant of $\text{NodeRank}_{\text{DeepCrime}}$, regardless of different effect size settings, exhibits higher effectiveness than that of $\text{NodeRank}_{\text{Random}}$. According to the experimental design mentioned above, if the effectiveness of $\text{NodeRank}_{\text{DeepCrime}}$ is higher than that of $\text{NodeRank}_{\text{Random}}$ and $\text{NodeRank}_{\text{withoutGMM}}$, we consider that the model mutations (generated by model mutation rules) in $\text{NodeRank}_{\text{DeepCrime}}$ contribute to mutated models. Therefore, the above experimental results indicate that the model mutation rule of NodeRank actually contributes to its effectiveness.

> **Answer to RQ5:** *The model mutation rules of NodeRank actually contribute to its effectiveness.*

4.5.6 RQ6: Influence of Mutation Operator Parameters on NodeRank

Objective: In NodeRank, we designed a set of new mutation operators specifically for GNNs. In this research question, we explore the influence of the parameter ranges of mutation operators on NodeRank.

Experimental design: First, we selected multiple mutation operators with parameters of integer/float types. This choice was made because, for Boolean-type mutation operators, mutations involve toggling between True and False, resulting in only one possible parameter value, rendering parameter changes unfeasible. Following the approach from the existing study [10], for each investigated mutation operator, we systematically varied its parameters multiple times while keeping the parameters of other mutation operators in their initial states. Subsequently, we recorded NodeRank's effectiveness (measured by APFD) after each parameter change. We used line graphs to visually depict the impact of parameter changes on NodeRank's effectiveness for each mutation operator.

Specifically, NodeRank consists of three types of mutation operators: Graph structure mutation (GSM), Node feature mutation (NFM), and GNN model mutation (GMM). Since these mutation operators aim to introduce subtle modifications to the original test set or the GNN model, we aim to ensure that after adjusting parameter ranges, the new parameter values also result in relatively slight changes.

- **Graph structure mutation (GSM)** GSM includes a mutation operator that involves slightly changing the structure of the input graph by randomly adding edges. Consequently, the parameter for this mutation operator is *the number of added edges*. This parameter was set to 1, 2, 3, and 4 to investigate the impact of the parameter range of this mutation operator on the effectiveness of NodeRank.
- **Node feature mutation (NFM)** NFM includes a mutation operator that changes the features of the targeted nodes to adjust their positions in the feature space. Consequently, the parameter for NFM is *the node feature offset*. This parameter was set to 0.05, 0.10, 0.15, and 0.20 to investigate the impact of the parameter range of this mutation operator on the effectiveness of NodeRank.
- **GNN model mutation (GMM)** GMM comprises multiple mutation operators

that aim to make slight changes to the training parameters in NodeRank. We selected mutation operators with integer/float-type parameters and adjusted their parameter ranges. These include "*Negative Slope*" with parameter range adjustments of (0.1, 0.2, 0.3, 0.4) and "*Hidden Channel*" with parameter ranges of (5, 10, 15, 20).

Results: The experimental results of RQ6 are presented in Figure 4.6. The experiments are conducted on 16 natural subjects. Among them, Figure 4.6a) shows the impact of changing the parameter number of edges for the mutation operator targeting graph structure. Figure 4.6b) illustrates the influence of the parameter node feature offset for the mutation operator targeting node features. Figure 4.6c) demonstrates the impact of the parameter negative slope for the mutation operator targeting the GNN models. Figure 4.6d) displays the influence of the parameter Hidden channel for the mutation operator targeting the GNN model. In this context, the red line represents NodeRank. First, we see that across all parameter settings of the mutation operator, NodeRank effectiveness consistently exceeds that of all the comparative test prioritization methods (i.e., GraphPrior, confidence-based approaches and random selection). Moreover, we found that NodeRank performs stably when the parameter values of the newly designed mutation operators change. For example, when modifying the "Number of edges" parameter, the APFD values of NodeRank vary within the range of approximately 0.778 to 0.785. Similarly, when adjusting the "Node feature offset" parameter, the APFD values of NodeRank fluctuate between approximately 0.777 and 0.785.

Answer to RQ6: *Across all parameter settings of the newly designed mutation operator, NodeRank's effectiveness consistently outperforms that of other comparative test prioritization methods. Moreover, the effectiveness of NodeRank remains stable when the parameter values change.*

4.6 Discussion

4.6.1 Generality of NodeRank

Our proposed NodeRank and its variants perform test prioritization for GNNs via ensemble-learning-based mutation analysis. The evaluation on 124 subjects demonstrates their effectiveness on both natural and adversarial datasets. The scheme of NodeRank, (i.e., slightly changing graph inputs and graph models) can also be generalized to edge-level and graph-level GNN tasks. In the future, we will carefully design relevant mutation rules to further adapt NodeRank to other GNN tasks.

Additionally, we discuss the potential applicability of NodeRank for regression tasks. However, currently, the mutation rules and ranking models of NodeRank are designed explicitly for classification tasks. To extend NodeRank to regression tasks, modifications to the model mutation rules and ranking models would be required. If appropriate model mutation rules can be identified for regression tasks and suitable ranking models can be designed, NodeRank could also be a promising approach for regression tasks.

4.6.2 Challenges of NodeRank

NodeRank requires a sufficiently large training set to train its internal ranking model. This training set includes labels (i.e., samples that the model predicts incorrectly are labeled as 1, while correctly predicted samples are labeled as 0). If the original model has very high accuracy, it can result in very few training samples labeled as 1, potentially leading to an imbalanced dataset during the training of NodeRank's ranking model. An imbalanced dataset can cause a decrease in performance when dealing with samples labeled as 1, as there are not enough examples to learn how to rank these samples correctly.

For example, in a scenario involving bank transfer transactions, where each account represents a node and edges represent transfer transactions between accounts, GNN models can be used to identify fraudulent accounts (i.e., whether a node is a fraudulent account or not). If the GNN model has a very high accuracy (few nodes predicted incorrectly), it will result in very few samples labeled as 1 in the NodeRank training set. This directly affects the training of the ranking model in NodeRank. Under these conditions, the effectiveness of NodeRank in prioritizing the misclassified accounts will be affected.

4.6.3 Differences in Approaches for NodeRank

In this Section, we discuss the differences in approaches for NodeRank from three perspectives, namely the differences in evaluating NodeRank methods, the differences between NodeRank and its variants, as well as different NodeRank approaches with different types of features.

[*Differences in evaluating NodeRank methods*] In addition to evaluating NodeRank on natural datasets, we assess its effectiveness from three different perspectives, as presented in RQ2 through RQ4. This is because these perspectives cover key aspects and contribute to a comprehensive understanding of NodeRank's performance. In RQ2, we assess the efficacy of NodeRank when confronted with adversarial test inputs. In RQ3, we explore how ensemble learning strategies influence NodeRank's effectiveness within the context of learning-to-rank. In RQ4, we examine the individual contributions of each category of mutation features (GSM, NFM, and GMM) that are generated for NodeRank's learning-to-rank model. Below, we provide a detailed explanation of the differences across approaches for RQ2, RQ3, and RQ4, as well as why it is important to assess the effectiveness of NodeRank from these three different perspectives.

- **RQ2 - Evaluation on Adversarial Test Inputs** This perspective focuses on assessing NodeRank's performance when confronted with adversarial test inputs. It is critical because it reveals NodeRank's resilience and reliability in handling challenging input data. In contrast to the evaluation methods in RQ2 and RQ3, this assessment is conducted using adversarial test inputs rather than natural datasets.
- **RQ3 - Impact of Ensemble Learning Strategies** This perspective investigates how different ensemble learning strategies influence NodeRank's effectiveness within the context of learning-to-rank. This investigation is significant as it helps us understand which strategies are more suitable for NodeRank to perform test prioritization.
- **RQ4 - Contributions of Mutation Features:** In RQ4, we delve into the individual contributions of each category of mutation features (GSM, NFM, and

GMM) on NodeRank. Understanding these differences is essential to identify which features are most critical for NodeRank's effectiveness, guiding further research and development efforts.

[*Differences between NodeRank and its variants*] In RQ3, we propose several variants of NodeRank. In RQ3, the variants of NodeRank differ in the ensemble learning strategies used to combine base ranking models. Apart from this distinction, the workflows of the NodeRank variants remain identical to NodeRank.

[*Different NodeRank approaches with different types of features*] In RQ4, we design different NodeRank approaches, which apply different types of mutation features for test prioritization. Specifically, NodeRank$_{NFM}$ only applies the NFM features. NodeRank$_{NFM+GSM}$ applies both the NFM and GSM features. Our aim is to investigate the contributions of each feature type to the effectiveness of NodeRank.

4.6.4 Threats to Validity

Threats to Internal Validity. The internal threat to validity mainly exists in the implementation of NodeRank, its variants, and the compared approaches. To reduce the threat, we implemented all approaches based on the widely used library PyTorch. Concerning the compared test prioritization approaches, we considered the implementations released by the authors. Another internal threat lies in the randomness of the model training process. To mitigate this threat, we conducted a statistical analysis involving performing ten repetitions of the model training process for both the original and mutated models. We then used these results to calculate the statistical significance of the experiments. The selection of the mutation rules used in our study represents another potential threat to the internal validity of our research. Despite our best efforts to identify model mutation rules, it is possible that there are other unknown training parameters that could serve as mutation rules. To mitigate this potential threat, we deliberately chose model mutation rules that could directly or indirectly impact node interdependence in the prediction process.
Threats to External Validity. The external threats to validity mainly stem from the selection of the graph datasets as well as the GNN models adopted for our study. This threat is mitigated by the diversity of the subjects, as well as by the fact that we consider assessing not only natural inputs but also adversarial inputs.
Threats to Construct Validity. Our mutation rules are similar to the attacks used under graph adversarial settings. This may, in theory, create a bias in the experimental results related to adversarial test input prioritization. However, this threat is mitigated by two elements: first, we also apply NodeRank on natural inputs; second, the objective of the mutation is eventually to generate features for learning to rank initial inputs, not generating new samples that will be part of the test suite.

4.7 Related Work

4.7.1 Test Prioritization Techniques

Test prioritization focuses on finding the ideal ordering of tests to detect more bugs in a limited time budget. In traditional software testing, a variety of approaches [11, 76, 77, 166, 167, 79] has been proposed. Mutation analysis has also been explored for test prioritization: Shin *et al.* [57] use a diversity-aware mutation adequacy criterion and demonstrate its effectiveness on large-scale developer-written test cases. Papadakis *et al.* [58] proposed mutating Combinatorial Interaction Testing models

for test prioritization. Gökçe *et al.* [168] introduced a prioritized testing approach aimed at enhancing the testing capacity of ESG-based testing algorithms. ESG-based algorithms, as discussed by *Belli et al.* [169], focus on generating software test suites that meet specific criteria related to both coverage and execution cost. Gökçe *et al.*'s approach leverages adaptive competitive learning algorithms for training the neural networks utilized in this process. The core objective of their work is to improve the test capacity of existing algorithms by prioritizing the testing process. GÖKÇE *et al.* [170] introduced a model-based approach to test prioritization. Their method focuses on providing an effective algorithm for ordering test cases based on the perceived degree of preference by the tester. Unlike code-based approaches, which rely on prior knowledge such as fault counts, or source code, GÖKÇE *et al.*'s approach is radically different. It does not require prior knowledge about the system under test (SUT), making it suitable for a wide range of testing scenarios.

For DNNs, Feng *et al.* [6] have proposed DeepGini, which prioritized test inputs based on model uncertainty: a test input is more likely to be incorrectly predicted if the DNN model outputs similar probabilities for different classes. PRIMA [10] is currently the state-of-the-art DNN test prioritization approach. It is based on intelligent mutation analysis guided by learning-to-rank. NodeRank shares similarities with PRIMA in the use of mutation analysis. Unfortunately, PRIMA's mutation rules are not applicable to GNNs and their inputs. Our work is thus the first approach that specifically leveraged mutation testing adapted to GNNs in order to achieve test input prioritization.

4.7.2 Mutation Testing

Mutation testing is commonplace in traditional software engineering [57, 58], where it constitutes a widely validated way to assess the quality of test cases. Mutation rules for traditional software have therefore been iteratively refined in the community. Recent studies have extended the applicability of mutation testing to various domains by focusing on adapting new mutation rules. Beyond simple bugs, Loise *et al.* [130] proposed 15 security-aware mutant operators to improve security testing. Beyond plain Java code, Deng *et al.* [171, 172] proposed novel mutant operators that are specifically designed to test Android applications (e.g., with event handling and activity lifecycle mutant operators).

Furthermore, in addition to the context of traditional software, several studies have investigated the application of mutation testing to DNNs and have proposed different mutation operators and frameworks. For instance, Ma *et al.* [20] proposed DeepMutation, a method to assess the quality of test data for DL systems using mutation testing. To achieve this, they designed a collection of source-level and model-level mutation operators to inject faults into the training data, programs, and DL models. The effectiveness of the test data is evaluated by analyzing the extent to which the injected faults can be detected. Later, Hu *et al.* [59] extended their work into a mutation testing tool for DL systems named DeepMutation++. This tool introduced new mutation operators for feed-forward neural networks (FNNs) and Recurrent Neural Networks (RNNs) and enabled the mutation of run-time states of an RNN. Another notable contribution is DeepCrime [60], a mutation testing tool that implements a set of DL mutation operators based on real DL faults. Shen *et al.* [61] proposed MuNN, a mutation analysis method for neural networks. MuNN defined five mutation operators based on the characteristics of neural networks.

4.7.3 Deep Neural Network Testing

In order to improve the test efficiency of DNNs, existing studies [6, 48, 46, 10, 173, 59, 47, 174] has proposed several approaches to optimize the test process, which is mainly divided into two categories. The first one is test input prioritization, which has been elaborated in the above section. The second one is test selection, which focuses on selecting a small group of test inputs to precisely estimate the accuracy of the whole testing set to reduce labelling costs. Li *et al.* [48] proposed Cross Entropy-based Sampling (CES) to select representative test inputs for DNN accuracy estimation, which minimizes the cross-entropy between the selected set and the entire test set to ensure the distribution of the selected test set similar to the original test set. Chen *et al.* [46] proposed PACE for test selection and accuracy estimation. Pace clusters all the inputs in a test set into different groups and leverages the MMD-critic algorithm [49] to select prototypes from each group. In addition to improving DNN testing efficiency, existing studies [51, 20, 8, 7, 52] have also focused on measuring DNN testing adequacy. Pei *et al.* [8] proposed neuron coverage to assess the extent to which a test set covers the DNN model logic. Ma *et al.* [7] proposed DeepGauge, a set of coverage-based metrics that consider neuron coverage a good indicator to evaluate the adequacy of test inputs. Kim *et al.* [52] proposed surprise adequacy, which assesses the adequacy of test inputs by measuring their surprise with respect to the training set.

4.8 Conclusion

To relieve the labelling-cost problem and improve the efficiency of GNN testing, we propose a novel test prioritization approach, NodeRank, which prioritizes test inputs that are more likely to be misclassified by the evaluated GNN model. NodeRank filled a gap in the literature: prioritization approaches that achieve state-of-the-art performance on DNNs are not suitable for GNNs since they ignore the interdependence between test inputs in graph-structured datasets. NodeRank leverages the concepts of mutation testing to perform test prioritization, with the aim of reducing the labelling cost in the process of evaluating a GNN model. Overall, NodeRank is a test prioritization approach that is model-based, input-based, and mutation testing-based. It utilizes mutation operations on both GNN models and test inputs to generate mutation features for each test input, facilitating test prioritization. The core idea is that: If a test input (node) can kill many mutated models and produce different prediction results with many mutated inputs, this input is considered more likely to be misclassified by the GNN model and should be prioritized higher. The specific process of NodeRank consists of two core steps: (1) NodeRank introduced three types of mutation rules to generate mutants from the perspective of the graph structure, node features, and the GNN model, respectively. (2) After obtaining the mutation results, NodeRank generated mutation feature vectors and utilized ensemble ranking models for test prioritization. Experimental results on 124 diverse subjects, considering natural and adversarial inputs, demonstrated the effectiveness of NodeRank. More specifically, NodeRank outperformed all the compared test prioritization approaches with an average improvement between 4.41% and 62.15%. Moreover, ablation experiments are performed to check that the different types of mutation features are all useful for the effectiveness of NodeRank.

Availability

Our replication package is available at

`https://zenodo.org/records/10049979`.

5 PriCod: Prioritizing Test Inputs for Compressed Deep Neural Networks

In this chapter, we propose PriCod, a novel test prioritization approach specifically designed for compressed DNN models. PriCod is rooted in two fundamental premises: firstly, that significant prediction deviations between compressed and original DNN models signify a greater likelihood of test input misclassification, and secondly, that test inputs situated near decision boundaries are more susceptible to misclassification. By prioritizing potentially misclassified test inputs, testers can allocate limited label budgets to these challenging inputs, thus speeding up the debugging process.

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

- Yinghua Li, Xueqi Dang, Jacques Klein, Yves LE Traon and Tegawendé F. Bissyandé. PriCod: Prioritizing Test Inputs for Compressed Deep Neural Networks. Under TOSEM major revision review.

Contents

5.1 Introduction

The widespread use of deep neural networks (DNNs) has brought significant advancements to machine learning in areas like computer vision [1, 2], autonomous vehicles [3, 4], and recommendation systems [5]. However, the increasingly complex DNNs require substantial computational resources and memory, limiting their practical deployment in resource-constrained environments, such as edge devices. To tackle these challenges, the research community has focused on the development of compressed DNN models that strike a balance between computational efficiency and model accuracy. Compressed DNN models, essentially scaled-down neural networks, are designed to sustain predictive accuracy while keeping computational requirements to a minimum. A multitude of compression techniques, including quantization [175], have emerged as valuable tools for reducing the size and computational load of DNNs while safeguarding their predictive prowess. Consequently, the evaluation and validation of compressed DNNs have become increasingly crucial to ensure their performance.

Existing studies [176, 177, 178] mentioned that, when evaluating compressed DNN models, labeling new test cases is necessary. However, labelling test inputs for compressed DNN models faces a central challenge: the high labelling cost issue. This challenge arises for two main reasons. Firstly, the scale of the test set can be extensive. Secondly, manual labeling is still the mainstream approach, requiring the participation of multiple annotators to guarantee the accuracy of the labeling process for each test input. A highly appealing solution to this challenge is test input prioritization. This technique prioritizes test inputs that are more likely to be misclassified when resources and time are limited for manual labeling. In contrast to traditional DNN models, test prioritization for compressed DNN models presents unique challenges: 1) Traditional DNN models usually consider two factors for test prioritization: the tests and the evaluated DNN model. For instance, DeepGini prioritizes tests by assessing the uncertainty level in the model's predictions for each test. However, test prioritization for compressed models can involve incorporating information from an additional source: the prediction deviation between the compressed DNN model and its original DNN model. Integrating data from this deviation information is also essential to improve the effectiveness of test prioritization. However, existing test prioritization methods for traditional DNNs do not take this important aspect into consideration; 2) In mutation-based test prioritization methods, the ranking of test cases is typically conducted by introducing model/input mutations, such as neuron effect block and weights shuffling [10]. However, due to the unique structure of the compressed model, some specific model mutation operations such as neuron activation inversion cannot be directly applied to compressed DNN models [19]. Below, we presented existing test prioritization approaches for traditional DNNs and discussed their limitations in the context of compressed DNNs. Specifically, current test prioritization methods can generally be categorized into three types: coverage-based, confidence-based, and mutation-based strategies.

Coverage-based approaches prioritize test inputs based on the extent of neuron coverage within DNNs. Conversely, confidence-based methods aim to identify potential misclassifications by quantifying the classifier's output confidence for each test case. Notably, the DeepGini approach by Feng et al. [6] utilizes the Gini score as a confidence metric for effective test prioritization. More recently, Weiss et al. [9]

conducted a comprehensive study, which included an evaluation of several confidence-based metrics, such as Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy. Mutation-based techniques propose various mutation operations and employ the modified results to prioritize test cases. While these methods have made progress in identifying potentially misclassified test inputs, they still face certain challenges and limitations when applied to compressed DNN models.

- Previous research [6] has demonstrated that coverage-based methods are less effective and more time-consuming when compared to confidence-based approaches.
- The mutation-based test prioritization approach, PRIMA [10], cannot be applied to compressed DNN models because the model mutation operators of PRIMA are not applicable to them. This limitation arises from the fact that the architectures and gradients of compressed models are typically unavailable [19].
- Furthermore, when employing confidence-based test prioritization approaches for compressed DNN models, these methods treat the compressed DNN models as black boxes and ignore the information regarding deviations before and after model compression when conducting test prioritization.

In this paper, we propose PriCod (**Pri**oritizing test inputs for **Co**mpressed **D**NN models), a test prioritization approach specifically tailored for compressed DNN models. PriCod leverages the deviation between the original model and compressed DNN models, along with the embedding information of test inputs, to perform test prioritization. The fundamental principle underlying PriCod is twofold:

- **Premise 1:** For a given test, if the prediction behavior between the compressed DNN model and the original model shows a large deviation, it suggests that, for this input, the compressed model is more likely to make a prediction different from that of the original model. This test is considered more likely to reveal bugs in the compressed model. We validated premise 1 through a targeted preliminary study. Details can be found in Section 5.3.1.
- **Premise 2:** Test inputs that are located closer to the decision boundary of the model are more likely to be misclassified. This premise has been previously established in prior research [20].

Building upon the aforementioned premises, PriCod generates two distinct types of features: deviation features and embedding features. To ensure a comprehensive representation of deviation information, PriCod employs a set of 17 strategies for generating deviation features. For each test input, PriCod combines these two feature types to predict the probability of the test being misclassified by compressed DNN models. Below, we provide an overview of the two types of features and elaborate on how PriCod utilizes them to achieve effective test prioritization.

- **Deviation Features** Deviation features are specifically designed to capture the impact of model compression on test inputs. They quantify the disparities in predictions between the original DNN model and the compressed DNN model for each test input. We generated 17 types of deviation features, such as Cosine Similarity [179] and Manhattan Distance [180], with the aim of providing a more comprehensive quantification of the disparity.
- **Embedding Features** Embedding Features encapsulate the representative information within each test input. Through the process of mapping a test input to a vector in space, PriCod aims to indirectly unveil the proximity between the test input and the decision boundary.

For each test instance, PriCod integrates the aforementioned two types of fea-

tures to derive the ultimate feature vector. Using this vector, PriCod learns the misclassification probability of this test. Ultimately, PriCod ranks all tests within a test set based on their misclassification probabilities.

Compared to test prioritization approaches for traditional DNNs, the unique novelty and contribution of PriCod lies in its tailored design for compressed models. Unlike traditional DNN test prioritization methods that rely on information from just the test set and the DNN model, PriCod incorporates a third key element: the original model from which the compressed model is derived. More specifically, PriCod utilizes the disparities between the original and compressed DNN models for test prioritization. By converting these differences into features, PriCod enhances the effectiveness of test prioritization in the context of compressed DNNs. The novelty of PriCod mainly exists in the following aspects.

- **Utilization of Deviation Features** PriCod first leverages the prediction deviation between the compressed DNN model and its original DNN model for test prioritization. To this end, PriCod introduces "deviation features" that aim to quantify the impact of model compression on each test input. Specifically, PriCod employs 12 different deviation evaluation metrics, aiming to comprehensively assess prediction deviation, thereby enhancing the effectiveness of test prioritization.
- **Integration of Embedding Features** In addition to the "deviation features", PriCod introduces "embedding features" for test prioritization, encompassing crucial information from each test input. By mapping test inputs to vectors in space, PriCod aims to indirectly reveal the proximity of test inputs to the model's decision boundaries, thereby guiding the test prioritization process.
- **Adopting Different Feature Fusion Strategies** PriCod employs various feature fusion strategies to combine the aforementioned deviation features and embedding features for test prioritization. According to the existing study [181], feature fusion can contribute to improving model predictive capability. By using different feature fusion techniques in PriCod, we can identify the most suitable strategy to enhance the effectiveness of PriCod's ranking model for test prioritization.

PriCod demonstrates its wide applicability across various contexts of compressed DNNs. For instance, in the context of medical image diagnosis, hospitals utilize DNN models to diagnose lung diseases in X-ray chest images. To overcome storage and computational limitations, they opt for compressed DNN models. However, this compression process carries the risk of accuracy loss, which can lead to treatment delays, missed early interventions, and patient health deterioration. PriCod can be used to effectively identify images at higher risk of misclassification by the compressed DNN model. By prioritizing these samples for screening, it can reduce the risk of misdiagnosis and enhance the reliability of the diagnostic model.

We conducted an extensive study to assess PriCod's performance, utilizing a dataset comprising 182 subjects (paired datasets and compressed DNN models). The evaluation covered a diverse range of test inputs, including natural data, noisy data, and adversarial data. Additionally, we meticulously selected a set of test prioritization approaches for comparison, which have previously proven effective in existing studies [9, 6]. Furthermore, we included random selection as the baseline approach. Our experimental results highlight PriCod's superior performance compared to existing methods. When applied to natural test inputs, PriCod demonstrates

an average improvement ranging from 7.43% to 55.89%. For noisy and adversarial test inputs, it exhibits an average improvement ranging from 7.92% to 52.91% and from 7.03% to 51.59%, respectively. We publish our dataset, results, and tools to the community on Github[1].

Our work has the following major contributions:

- **Approach** We propose PriCod, a novel test prioritization approach designed specifically for compressed DNN models. Our approach leverages the discrepancies in predictions before and after model compression, as well as the embedding features of tests, to guide test prioritization.
- **Study** To evaluate PriCod, we conduct an extensive study involving 182 subjects, encompassing natural, noisy, and adversarial datasets. Within this study, we systematically evaluate PriCod in comparison to multiple test prioritization approaches. Our experimental results demonstrate the effectiveness of PriCod.
- **Performance Analysis** We assess the contributions of various feature types to the performance of PriCod. Additionally, we analyze the impact of feature fusion techniques on PriCod's effectiveness. Furthermore, we investigate the relationship between the misclassification probability and the deviated behaviors.

5.2 Background

5.2.1 DNNs and DNN testing

Classification deep neural networks (DNNs) [87] serve as the core of numerous deep learning (DL) systems. Classification refers to the task of categorizing input data into different classes [182], such as identifying objects in images or categorizing emails as spam or non-spam. DNNs can perform the classification task by learning mappings from input data to specific categories. They adjust the weights and parameters within the network through training data, enabling the network to automatically capture patterns and features in the data, thereby achieving accurate classification.

A DNN consists of multiple layers: an input layer, an output layer, and one or more hidden layers. Each layer is composed of a series of neurons. The input layer is the first layer of the network, responsible for receiving raw data. The output layer is the final layer of the network, generating the final prediction results. Hidden layers lie between the input and output layers. They transmit information and perform feature extraction within the network. Each hidden layer comprises multiple neurons that combine and transmit information through weights and activation functions. In the context of DNNs, neurons are the fundamental computational units. Neurons in hidden and output layers are interconnected with all neurons in the preceding layer through weighted edges. The weights of these edges are automatically learned during a training process using a large dataset with labeled training examples. Following training, a DNN can autonomously classify input samples, such as images, into their respective categories. For example, within the framework of a DNN model designed for animal classification tasks, it can differentiate between various types of animals in images, precisely labeling whether it is a cat, dog, or bird.

Ensuring the quality and reliability of DNN models is of paramount importance. DNN testing has emerged as a widely adopted approach to achieve this goal [111, 60, 108, 183]. Similar to traditional software systems [77, 184, 185, 78], DNN testing relies on the use of inputs and oracles. In the context of DNN testing, test inputs

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

represent the data that the model is expected to classify. These inputs can take various forms depending on the specific task of the DNN under examination, such as images, natural language, or speech. Test oracles in DNN testing involve manual labeling, wherein human annotators manually assign ground truth labels to each input. By comparing these labeled ground truth labels with the predicted output of the DNN model, it becomes possible to evaluate the model's accuracy in generating the correct output for a given input.

5.2.2 DNN Model Compression

Model compression has emerged as a promising avenue of research to facilitate the deployment of DNN models [28, 29, 30]. The primary objective of DNN model compression is to minimize the computational and memory requirements of models while maintaining their performance, thus enabling effective deployment in resource-constrained contexts. A variety of techniques have been proposed for compressing DNN models. Among these techniques, Quantization plays a crucial role, which operates by compressing a DNN model through the adjustment of bit numbers allocated to weight representation [31]. In the conventional landscape of DNN models, weights find their common expression in the form of 32-bit floating-point numbers. The primary goal of quantization is to reduce the bit precision of parameters in neural network models, thereby decreasing storage and computational overhead while maintaining optimal model performance. This involves the utilization of reduced bit representations like 8-bit, which in turn substantially curtails the storage requirements of the model. In our study, we primarily employed quantization as the method for model compression.

The currently trending model compression techniques primarily encompass two options: TensorFlow Lite (TFLite) [32] and CoreML [33]. These two compression methodologies have gained extensive traction within the mobile device domain [70]. TFLite, developed by Google, is a deep learning inference framework explicitly designed for mobile and embedded devices. Its standout feature is its highly optimized computational performance, which ensures the efficient execution of trained neural network models on the Android platform [34]. On the other hand, CoreML is Apple's solution for deep learning inference on mobile devices, tailored exclusively for the iOS ecosystem. CoreML leverages the inherent hardware advantages of Apple devices and achieves rapid inference for neural network models through strategic hardware acceleration implementation. In our research, we employed the aforementioned techniques (TFLite and CoreML) to compress the original DNN models into compressed DNN models, aiming to offer a more comprehensive evaluation.

5.2.3 Confidence-based Test Prioritization for DNNs

Confidence-based methods identify inputs that can potentially expose bugs (i.e., possibly misclassified test inputs) by analyzing the output probabilities of a DNN classifier. One classic confidence-based test prioritization technique is DeepGini, which prioritizes test inputs by calculating Gini scores for each input. These scores measure the model's confidence in classifying each input, thus facilitating test prioritization. More specifically, if a DNN produces similar probabilities for all classes towards a test input, it indicates lower confidence in the classification. As a result, this input will be prioritized higher. DeepGini has demonstrated effectiveness across

various prevalent DNN datasets, such as CIFAR10 (color images) [186] and Fashion (fashion product images) [187]. Recently, Weiss *et al.* [9] extensively explored diverse techniques for prioritizing DNN test inputs, encompassing a series of confidence-based approaches, such as Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy. These metrics have been proven effective in DNN test prioritization. Compared with coverage-based test prioritization methods (e.g., CTM and CAM), which prioritize inputs based on neuron coverage, confidence-based methods offer several distinct advantages.

- **Efficiency** Confidence-based methods require minimal time and computational resources, as they solely rely on statistical computations of confidence levels in the predicted probability vectors from the output softmax layer.
- **Effectiveness** Confidence-based methods have demonstrated higher effectiveness compared to various coverage-based test prioritization techniques.
- **Minimal Need for Intermediate Information** Unlike coverage-based methods, confidence-based approaches do not necessitate the collection of extensive intermediate information to compute coverage rates. Additionally, they enhance security by not requiring an in-depth inspection of the DNN, thereby safeguarding sensitive information embedded within the network.

However, applying confidence-based methods to compressed DNN models comes with a limitation. These methods solely depend on the predictive confidence information of the compressed DNN model without considering the differences between the compressed model and the original model. Our proposed approach, PriCod, incorporates this deviation information in the test prioritization process. We compared PriCod with a set of confidence-based test prioritization techniques and demonstrated that Pricod outperforms all the comparative methods, as evidenced by Section 6.5.

5.3 Approach

5.3.1 Preliminary Study

The core idea of premise 1 is that, given a test input, if there is a large deviation in the prediction behavior between the compressed DNN model and the original model, it indicates that this test have a high probability to be misclassified by the compressed DNN model. In this section, to validate the rationality of Premise 1, we conducted the following preliminary study:

Objectives: We investigate the relationship between the prediction deviation resulting from model compression and the probability of the test being misclassified by the compressed DNN model.

Experimental design: We utilized a variety of distance measurement metrics, such as Euclidean Distance and Manhattan Distance, to investigate the correlation between prediction deviations and misclassification probabilities. For each distance metric, we evaluated the prediction deviations between the original DNN model and the compressed model for each sample in the test dataset, accordingly assigning a deviation score. Subsequently, we ranked all the test samples in descending order based on their deviation scores. We then divided the samples into ten equally sized groups, with deviations progressively decreasing across these ten segments. Within each segment, we identify the number of misclassified tests to observe whether there was a relationship between prediction deviation magnitude and the likelihood of misclassification.

Results: The experimental results of the preliminary study are presented in Table 5.1 and Figure 5.1. Table 5.1 displays the number of misclassified tests in different deviation levels of test groups based on various distance metrics. The "Deviated Behavior Metrics" in the table represent the metrics used to measure deviation. Furthermore, for each metric, we sorted all tests in the test set according to the magnitude of their deviation behavior. The range of 0%-10% indicates the top 10% of samples with the highest deviations. Similarly, 10% to 20% represents samples with deviation magnitudes falling within the top 10% to 20% interval. In each test group (i.e., 10%~20% and 20%~30%), the total number of samples is the same.

Table 5.1: Correlation between prediction deviation in the original model and the compressed model and misclassification of tests

Deviated Behavior Metrics	**Percentage of tests selected**									
	0%-10%	10%-20%	20%-30%	30%-40%	40%-50%	50%-60%	60%-70%	70%-80%	80%-90%	90%-100%
Euclidean	877	815	769	705	635	499	347	200	81	28
Manhattan	918	851	794	712	618	475	309	174	77	26
Chebyshev	868	811	755	690	628	504	356	230	86	27
SSD	877	815	769	705	635	499	347	200	81	28
Wasserstein	859	826	789	725	633	498	338	183	78	27

To provide a more precise illustration of the data in the table, we provide a concrete example: the number in the first row and first column of the table indicates that, when using the Euclidean method as the deviation metric, among the top 10% of tests with the highest deviation between the original model and the compressed model, there were 877 tests misclassified by the compressed model. It is important to note that the experiments for this research question were conducted using natural datasets, and the number of misclassified tests represents the mean across all 20 subjects.

We see that, as the deviation level decreases, regardless of the distance metric used to measure the deviation, the number of misclassified tests in each group decreases. For instance, with the Manhattan metric, in the top 10% of tests with the highest deviation, there were 918 misclassified tests. In the 10% to 20% deviation range, there were 851 misclassified tests, and this number decreased to 794 in the 20% to 30% range and further to 712 in the 30% to 40% range. Only 26 tests were misclassified in the 90% to 100% range. To visually represent this decreasing trend, we provide Figure 5.1, where each curve represents the relationship between deviation and misclassification for different distance metrics. In Figure 5.1, we see that as the deviation level decreases, the number of misclassified tests in the test groups gradually decreases. The above experimental results indicate that for a given test, if the original DNN model and the compressed model exhibit higher deviation in their predictions, the test is more likely to be misclassified by the compressed model.

Finding Tests with higher prediction disparities between the original DNN model and the compressed model are more likely to be misclassified by the compressed model.

5.3.2 Overview of PriCod

In this paper, we introduce PriCod, a novel test prioritization approach specifically designed for compressed Deep Neural Network (DNN) models. The overview of PriCod is depicted in Figure 5.2. Overall, the workflow of PriCod consists of four steps: Deviation Feature Generation, Embedding Feature Generation, Feature Fusion, and Feature-based Ranking. We provide a brief overview of these four steps below.

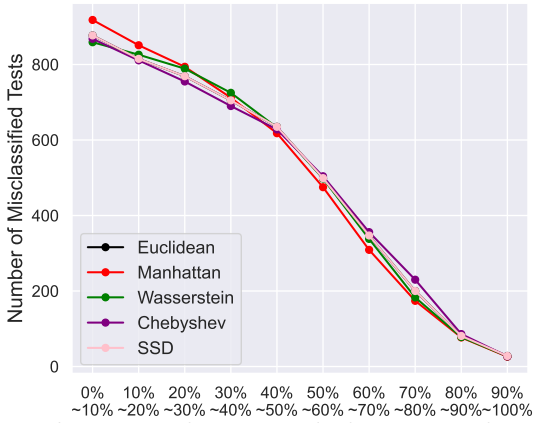


Figure 5.1: Correlation between deviation behavior and misclassification of tests. X-Axis: Tests sorted by decreasing deviation; Y-Axis: the number of misclassified tests

For detailed explanations, please refer to Section 5.3.3 to Section 5.3.6.

❶ **Deviation Feature Generation** PriCod initially generates deviation features for each test input. These features capture the behavioral differences between the compressed DNN model and its original DNN model when predicting a given test input t. The utilization of deviation features for test prioritization is grounded in the premise that, for a given test, if the prediction behavior between the compressed DNN model and the original model exhibits a large deviation, this test is considered more likely to be misclassified by the compressed DNN model. (The preliminary study conducted in Section 5.3.1 further validates this premise).

❷ **Embedding Feature Generation** For each test case, PriCod also generates embedding features to reflect the key characteristics of the test (image/text). This step is based on the premise that test inputs closer to the model's decision boundary are more likely to be misclassified. By mapping each test input to a vector in space, the embedding features can indirectly indicate the proximity of the test to the decision boundary.

❸ **Feature Fusion** PriCod integrates deviation features and embedding features, generating a more comprehensive feature representation for each test input.

❹ **Feature-based Ranking** Utilizing the generated fused feature vector, PriCod employs the LightGBM model to calculate misclassification scores for each test input. A high score for a test implies that it is more likely to be misclassified by the compressed model. Therefore, PriCod ranks all tests in descending order based on the misclassification scores.

Below, we explain the rationale for the design.

- **Deviation feature generation - prioritizing tests based on behavioral deviation:** A key insight of PriCod is that if there is a high prediction deviation between the compressed model and the original model for a given test, it is highly likely that the compressed model has potential issues arising from the compression when handling this test. Therefore, PriCod utilizes the deviation information to perform test prioritization. The feasibility of this approach is demonstrated by the preliminary study conducted in Section 5.3.1.
- **Embedding feature generation - prioritizing tests based on proximity to the decision boundary:** An existing study [20] has pointed out that test inputs close to the decision boundary are more likely to be misclassified. PriCod generates embedding features to indirectly reflect the proximity of each test input

to the decision boundary, thus performing test prioritization.

- **Feature fusion - enhancing the predictive power of the ranking model:** According to the existing study [181], feature fusion can enhance the predictive capabilities of models. Through feature fusion, we aim to enhance the ability of PriCod's ranking model to predict misclassification scores for each test, thereby improving the effectiveness of test prioritization.
- **Feature-based ranking - predicting the misclassification probability based on features:** Given a test input, PriCod utilizes the LightGBM model to estimate the probability of it being misclassified based on the fused features. LightGBM has been proven to be a powerful algorithm capable of predicting the probability values for different categories [89]. Within the framework of PriCod, LightGBM categorizes tests into two groups: those "misclassified by the compressed model" and those "not misclassified by the compressed model". We utilize LightGBM to predict the probability of each sample being misclassified by the compressed model, referred to as the misclassification score. A high misclassification score implies that the test is more likely to be misclassified by the compressed model. Therefore, PriCod ranks all tests in descending order based on the misclassification scores.

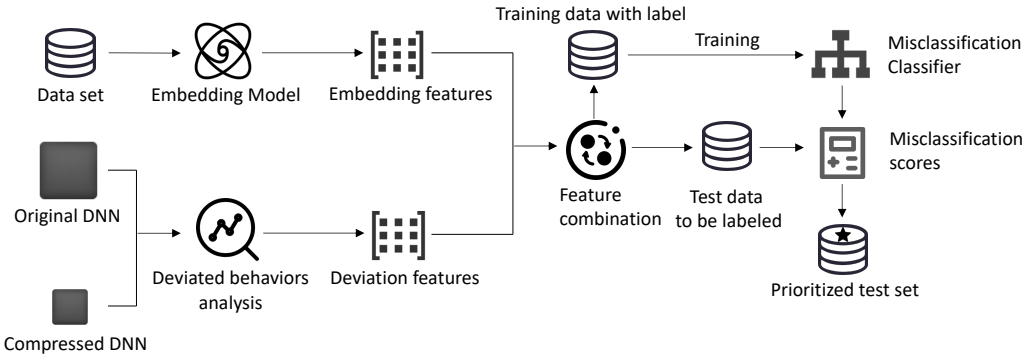


Figure 5.2: Overview of PriCod

5.3.3 Deviation Features Generation

During the process of model compression, which aims at reducing storage and computational costs, the model can lose some details and complexity, resulting in a relatively simplified compressed model. Deviations in prediction behavior can arise between the original DNN model and the compressed DNN model. Our core premise for prioritizing testing based on these behavioral discrepancies is as follows: When there is a significant deviated behavior between the compressed DNN model and the original model for a given test, it suggests that the compressed model is more likely to produce a prediction different from that of the original model for this input. This test is deemed more likely to expose bugs in the compressed model. We validated premise 1 through a specially designed preliminary study. Further details can be found in Section 5.3.1.

In order to quantify the magnitude of differences in predictions between the original model and the compressed DNN model, we propose 17 strategies for generating deviation features, with the aim of effectively encapsulating variations in predictions between the original model and the compressed DNN model. Our objective is to provide a comprehensive suite of measures to capture different aspects of deviation before and after model compression for each test input.

- **Classification Deviation Features (CLA)** [19]: These features capture the

disparities in predicted classes between the original DNN model and the compressed DNN model for a given test input. To derive these features, we compare the specific category predictions for each test case obtained from the original DNN model and the compressed DNN model. When the predictions differ, it signifies that the compressed model displays classification behavior contrary to that of the original model, providing a direct reflection of the variations in predictive behavior.

- **Confidence Deviation (CFD)** [6]: CFD reflects the absolute difference between the probability of the predicted category by the original model and the probability of the same category by the compressed DNN model. It reflects the degree of variation in the models' confidence levels.
- **Euclidean Distance Features (EUCL)** [83]: These features measure the Euclidean distance between the probability vectors of predictions made by the original model and the compressed model for a given test input. This distance metric reflects the overall magnitude of differences between the models' prediction probabilities, providing a comprehensive view of their predictive disparities.
- **Manhattan Distance Features (MHD)** [84]: MHD represents the Manhattan Distance between the probability vectors of predictions made by the original model and the compressed model for a given test input. It quantifies the sum of absolute differences between corresponding probabilities, indicating the overall shift in predictions and the directions of these shifts.
- **Chebyshev Distance Features (CHD)** [188]: CHD reflects the Chebyshev Distance between the probability vectors of predictions made by the original model and the compressed model for a specific test input. This metric highlights the maximum absolute difference between corresponding probabilities, showcasing the most significant deviations in the models' predictions.
- **Pearson Correlation Coefficient Features (PCC)** [86]: PCC measures the Pearson Correlation Coefficient between the probability vectors of predictions made by the original model and the compressed model for a given test input. It indicates the strength and direction of a linear relationship between the predictions, offering insights into the consistency of their deviations. A higher PCC value indicates smaller disparities in predictive behavior between the original model and the compressed model, while a lower PCC value indicates relatively larger disparities.
- **Sum of Squared Differences Features (SSD)** [85]: SSD reflects the Sum of Squared Differences between the probability vectors of predictions made by the original model and the compressed model for a specific test input. It emphasizes larger deviations while downplaying smaller ones, providing a measure of the overall prediction divergence.
- **Hellinger Distance Features (HED)** [189]: HED reflects the Hellinger Distance between the probability vectors of predictions made by the original model and the compressed model for a given test input. It measures the similarity between the square root of the prediction probabilities, offering insights into their predictive differences.
- **Wasserstein Distance Features (WAS)** [190]: WAS reflects the Wasserstein Distance between the probability vectors of predictions made by the original model and the compressed model for a specific test input. It measures the minimum "cost" of transforming one distribution into another, highlighting their divergence.

- **Coordinate Deviation Features**: CDF is obtained by subtracting the origin coordinates from the prediction vector of the compressed model. This vector directly reflects the deviation in predictions of the compressed DNN models from the origin coordinates.
- **Relative Entropy Features (REL)** [191]: REL reflects the Relative Entropy between the probability vectors of predictions made by the original model and the compressed model for a given test input. It measures the information lost between the compressed model and the original model.
- **Difference Vector Features (DIF)**: DIF refers to the vector obtained by subtracting the prediction vector of the original model from the prediction vector of the compressed model for a specific test input. This vector directly reflects the direction and magnitude of deviation in predictions.

The aforementioned deviation features exhibit the following differences:

1) Classification Deviation Features (CLA) vs. Confidence Deviation (CFD) CLA focuses on the difference in classification results, i.e., whether the compressed model produces the same output classification for a given test as its original model. CFD measures the deviation in confidence, indicating the differences in prediction uncertainty between a compressed model and its original model for a given test.

2) Euclidean Distance (EUCL) vs. Manhattan Distance (MHD) EUCL measures the straight-line distance between the prediction vectors of the compressed DNN model and its original DNN model, making it suitable for quantifying overall deviations in predictions in a multi-dimensional space. On the other hand, MHD calculates the sum of absolute differences in each dimension based on the prediction vectors.

3) Chebyshev Distance (CHD) vs. Sum of Squared Differences (SSD) CHD emphasizes the difference in the maximum single dimension of the prediction vectors, particularly highlighting extreme values. SSD represents the sum of squared differences across dimensions, offering a quantitative measure of overall deviation.

4) Hellinger Distance (HED) vs. Wasserstein Distance (WAS) HED measures the disparities in the shape of distributions between the prediction vectors generated by the compressed model and those of its original model for a specific test. In contrast, WAS focuses on the "effort" required to transform one distribution into another.

5) Relative Entropy (REL) vs. Pearson Correlation Coefficient (PCC) REL specifically focuses on information loss or gain to quantify the disparity in prediction distributions between the compressed model and its original model. On the other hand, PCC measures the degree of linear correlation among the prediction vectors.

6) Coordinate Deviation Features (CDF) vs. Difference Vector Features (DIF) Given a test input, CDF reflects the absolute prediction bias between predictions made by the compressed model and those by the original model. On the other hand, DIF emphasizes changes in both direction and magnitude relative to the predictions of the original model and the compressed model.

The aforementioned deviation features have proven to be useful in the context of PriCod for test prioritization, as evidenced by the preliminary study conducted in Section 5.3.1 and RQ5 (Feature contribution analysis). Specifically, the rationale behind these features is derived from PriCod's premise 1, which states that for a

given test, if the prediction behavior between the compressed DNN model and the original model exhibits a large deviation, it suggests that this test is more likely to be misclassified by the compressed DNN model. The aforementioned features are all designed to measure the prediction differences. Therefore, these features can be utilized for test prioritization. The validation of premise 1's reasonability is established in the preliminary study (cf. Section 5.3.1), and the effectiveness of deviation features is further demonstrated in RQ5 (cf. Section 5.5.5), which leverage ablation studies to confirm the contributions of the deviation features.

5.3.4 Embedding Features Generation

Embedding Features (EF) capture the intrinsic information of each test input $t \in T$. In our experiments, PriCod was evaluated under two scenarios: image-type tests and text-type tests. In the following sections, we present the generation process of embedding features under each of the scenarios. In the context of image-type tests, to obtain these Embedding Features (EFs), we utilize a pre-trained ResNet model [87] to transform each test into a vector representation. In the case of text-type tests, we employ the BERT [192] model to map each input into a corresponding vector representation.

The process by which ResNet transforms an image into an embedding feature vector is as follows: First, the pre-trained ResNet network is employed to load the image. Through successive layers of convolution and pooling, the image undergoes a gradual transformation into higher-level abstract features. The output of the final global average pooling layer is extracted and treated as the image's embedding. This embedding encapsulates the semantic information of the image.

The principle behind prioritizing testing using the Embedding Features of test cases is that: *test inputs closer to the decision boundary of the model are more likely to be misclassified*, as outlined in existing literature [20]. By mapping images to vectors in space, PriCod can automatically learn the distance between a test and the decision boundary to perform effective test prioritization. The benefits of utilizing the ResNet model to generate embedding features are outlined below:

- **Automatic Identification of Vital Features** The ResNet model can automatically extract crucial features from raw data. By generating embedded features, key characteristics can be focused.
- **Effective Feature Generation** ResNet boasts powerful feature generation capabilities. Its architecture integrates multiple convolutional and pooling layers, allowing the model to effectively capture a wide range of features and patterns within images. This capability can be highly advantageous for test prioritization.

The process by which BERT converts textual data into an embedding feature vector can be described in the following steps:

- **Tokenization** The input text, whether a sentence or a paragraph, is broken down into smaller units called tokens. These tokens are typically words or subwords, and each is assigned a unique identifier.
- **Embedding** Each token identifier is transformed into a corresponding word vector. These vectors are rich in semantic information, representing not just the token but also aspects of its meaning.
- **Contextual Analysis with Transformer Encoders** BERT utilizes multiple layers of Transformer encoders to process these word vectors. These encoders are adept at capturing contextual information, allowing the model to understand the

nuances and varied meanings of words based on their context in the sentence or paragraph.

- **Generation of Embedding Vectors** The final hidden states outputted by the Transformer encoders serve as the embedding vectors. These vectors represent the entire input text (sentence or paragraph) and encapsulate both the semantic and contextual information of the original text.

The advantages of the BERT model include: BERT takes into account contextual information when processing text, and the vectors it generates can better represent the semantic meaning of the text. As a result, when text information is mapped into space, BERT can place semantically similar texts closer together in this space, with texts of the same category being nearer to each other and different texts being farther apart. These vectors in space can indirectly reflect the distance from the decision boundary.

5.3.5 Feature Fusion

Building upon the above procedures, PriCod produces two distinct categories of feature vectors for each test sample in T: deviation feature vector and embedding feature vector. Following this, for each test sample $t \in T$, PriCod combines the two feature vectors and input to the LightGBM classifier, facilitating the prediction of the misclassification score for the test case. The process of feature fusion serves to enhance the effectiveness of subsequent test prioritization. Each type of feature (deviation features, embedding features) captures distinct aspects of the data, which hold informative value for the final prioritization. Through the amalgamation of these features into a singular feature vector, multiple sources of information are effectively integrated.

5.3.6 Feature-based Ranking

After obtaining the feature vector for each test t in set T, PriCod employs the **LightGBM classifier** [89] as the ranking model to predict the misclassification probability for each t based on its corresponding feature vector. LightGBM is a gradient-boosting framework that employs decision trees as base learners. LightGBM is designed for efficient parallel computation. In the subsequent section, we elaborate on the procedures of constructing the classifier and elucidate the adaptations carried out on the classifier to generate the misclassification scores instead of producing categories.

- **Construction of the LightGBM Classifier** Given the compressed DL model M and the evaluated dataset, to build the classifier, we first partitioned the dataset into two sets: a training set labeled as R and a test set labeled as T, with a partition ratio of 7:3 [92]. The test set T remains untouched for evaluating PriCod. The LightGBM ranking model is trained using the dataset R', which is derived from the original training set R of the evaluated compressed model. The process of constructing the training set for the ranking model R' is described below. Initially, we generate deviation features and embedding features for each instance $r \in R$. These features serve as the training features for the new dataset R'. Subsequently, we construct the labels for R'. To this end, we employ the compressed DNN model M to predict the classification of each instance $r \in R$. We compare the model's predictions with the corresponding ground truth of r to determine whether r is misclassified. Instances that are misclassified are labeled

as 1, while correctly classified instances are labeled as 0. Consequently, we obtain the labels for the training set of the ranking model. By utilizing the constructed training set, we train the ranking model LightGBM.

- **Adapting the LightGBM Classifier** Once the training is complete, LightGBM can be used to predict whether a test input will be misclassified by the compressed DNN model. To enable the model to produce misclassification scores as outputs instead of labels, we introduced specific modifications to the original LightGBM classifier. More precisely, we extract the intermediate value from the model's output, which initially served to determine whether a test would be misclassified by the model. In the model's decision-making process, if this intermediate value exceeds a certain threshold, the input is classified as "misclassified"; otherwise, it is classified as "not misclassified." Within the adjusted LightGBM classifier, this intermediate value refers to the misclassification score. A higher value signifies that a test instance is more likely to be misclassified.
- **Test Prioritization** Ultimately, PriCod ranks all the tests within the test set T in a descending order based on their respective misclassification probability scores. This sorting procedure yields the prioritized test set denoted as T'.

In the above steps of constructing the LightGBM classifier, we utilized deviation features and embedding features as training features to train the LightGBM model. We selected these specific features due to their ability to reflect the probability of a test being misclassified by a compressed DNN model. The rationale behind selecting these features is grounded in the core premises of PriCod (cf. Section 5.3.3 and Section 5.3.4):

- **Premise 1 (for deviation features)** For a given test, if the prediction behavior between the compressed DNN model and the original model exhibits a large deviation, it suggests that this test is more likely to be misclassified by the compressed DNN model. This premise indicates that deviation features can reflect the probability of a test being misclassified. The preliminary study conducted in Section 5.3.1 further validates this premise.
- **Premise 2 (for embedding features)** Test inputs that are located closer to the decision boundary of the model are more likely to be misclassified [20]. This premise elucidates that embedding features can reflect the probability of a test being misclassified. Specifically, by mapping each test input to a vector in space, the embedding features can indirectly indicate the proximity of the test to the decision boundary.

In the training set, each training sample is labeled as 0 or 1. Specifically, 0 means the sample was not misclassified by the compressed DNN model under evaluation, while 1 indicates the sample was misclassified. PriCod first generates deviation features and embedding features from each training sample. Using the LightGBM model, it learns the relationship between these features and "being misclassified". Following the completion of training, given a new test input, PriCod can quantify the probability of this test being misclassified based on its deviation features and embedding features.

5.3.7 Variants of PriCod

To comprehensively investigate how different feature fusion methods affect the overall performance of PriCod, we propose three distinct PriCod variants. These variants were meticulously designed to incorporate different approaches for combining

embedding features and deviation features, shedding light on the impact of feature fusion techniques on PriCod's effectiveness. These three variants are denoted as PriCod^a, PriCod^m, and PriCod^c, each adopting a unique strategy for feature fusion: addition, multiplication, and cross-multiplication [193], respectively.

- **PriCod^a** The variant PriCod^a utilizes an addition-based fusion method for test prioritization. In this variant, the feature fusion strategy relies on addition, combining the embedding features and deviation features by adding them. Addition-based fusion is straightforward and results in a merged feature vector where the corresponding elements of the two input feature vectors are summed together.
- **PriCod^m** The variant PriCod^m employs a multiplication-based fusion method for test prioritization. This approach involves multiplying each element of the embedding features by the corresponding element of the deviation features. Multiplication-based fusion is more intricate compared to addition-based fusion and has the potential to emphasize interactions and relationships between the two feature sets.
- **PriCod^c** PriCod^c employs a cross-multiplication feature fusion strategy. This approach performs cross-multiplication on elements of the embedding features and deviation features. Cross-multiplication is a more complex fusion method compared to both addition and multiplication. It allows the model to capture intricate interdependencies between features by considering all possible pairwise interactions.

To enhance the clarity of each feature fusion strategy, we incorporate an illustrative example. Given a test t, we first follow the steps outlined in Section 5.3.3 and Section 5.3.4 to obtain its embedding feature vector and deviation feature vector. Assuming these two vectors are $\{e1, e2, e3\}$ (embedding features) and $\{d1, d2, d3\}$ (deviation features), we present the final vectors obtained after addition-based fusion, multiplication-based fusion, and cross-multiplication-based fusion below. **Addition-based fusion:** $\{e1 + d1, e2 + d2, e3 + d3\}$. **Multiplication-based fusion:** $\{e1 \times d1, e2 \times d2, e3 \times d3\}$. **Cross-multiplication-based fusion:** $\{e1 \times d1, e1 \times d2, e1 \times d3, e2 \times d1, e2 \times d2, e2 \times d3, e3 \times d1, e3 \times d2, e3 \times e3\}$. In the following, we explain the rationale behind the studied feature fusion strategies.

- **Feature fusion based on addition (PriCod^a)** The rationale behind this fusion approach mainly consists of three points. 1) This approach is intuitive, simply adding the information of two feature sets together, making it easy to understand and implement. 2) by simple addition, this method preserves the complete information of each feature set. 3) This approach is widely used in the context of DNNs and has been proven to be effective [87].
- **Feature fusion based on multiplication (PriCod^m)** The rationale behind the multiplication-based fusion approach mainly consists of three points. 1) Multiplication-based fusion emphasizes the interdependence between features. 2) Multiplying certain input elements by smaller weights can contribute to ignoring irrelevant information.
- **Feature fusion based on cross-multiplication (PriCod^c)** 1) The cross-multiplication fusion allows the model to capture more complex interdependencies and interactions between features by considering possible pairwise interactions. 2) This approach creates a high-dimensional feature space capable of revealing more hidden patterns and relationships. 3) Existing research [193] has proven the effectiveness of this method.

While we have introduced diversity in the feature fusion aspect, we intentionally maintained all other aspects of the PriCod variants identical to the original PriCod, aiming to ensure that any observed performance differences can be attributed primarily to the selected fusion method.

5.4 Study design

In this section, we present a comprehensive elucidation of the specific details concerning our study design. To begin, Section 5.4.1 elucidates the research questions that guided our investigation. Subsequently, Section 5.4.2 provides intricate insights into the compressed models and datasets adopted in our study. Section 5.4.3 showcases the noisy generation techniques utilized in RQ2, while Section 5.4.4 exhibits the adversarial attacks employed in the context of RQ3. Moreover, Section 5.4.5 demonstrates the test prioritization methods subjected to comparison. In addition, Section 5.4.6 outlines the measurement metrics we employed to assess the effectiveness of PriCod, its variants, and the compared approaches. Finally, Section 5.4.7 provides a comprehensive overview of the implementation and configuration setup utilized throughout our study.

5.4.1 Research Questions

Our experimental evaluation answers the research questions below.

- **RQ1: How does PriCod perform in prioritizing test inputs for compressed DNN models?**
  When utilizing confidence-based test prioritization techniques for compressed DNN models, these methods typically treat the compressed DNN models as black boxes, neglecting valuable information about deviations before and after model compression in the prioritization process. In our research, we introduce PriCod, a tailored test prioritization approach explicitly designed for compressed DNNs. PriCod harnesses prediction disparities induced by model compression, in combination with test input embeddings, to efficiently prioritize tests that may reveal misclassifications. In this study, we assess the effectiveness of PriCod by comparing it to a set of existing test prioritization methods.
- **RQ2: How does PriCod perform on noisy test inputs?**
  When implemented on mobile devices, compressed DNN models can encounter noisy data due to a range of factors, such as capturing photos from various angles, as well as the presence of raindrops. In this research inquiry, we employ a set of noise generation techniques [93, 94, 95, 96] to construct noisy datasets for evaluating the performance of PriCod.
- **RQ3: How does PriCod perform on adversarial inputs?**
  The previous research questions have assessed PriCod's effectiveness with natural and noisy test inputs. In this research question, we evaluate PriCod's performance with adversarial test inputs.
- **RQ4: How does the feature combination strategies impact the effectiveness of PriCod?**
  In this research question, we aim to gain a deeper understanding of the impact of different feature combination strategies on the effectiveness of PriCod. By analyzing this critical factor, we can determine which feature fusion technique is better suited for PriCod.
- **RQ5: To what extent does each type of features contribute to the**

Table 5.2: Compressed DNN models and datasets

ID	Dataset	# Size	Model	Compression Tool	Supported Mobile Platforms
1	CIFAR10	60,000	AlexNet-coreml	Core ML	iOS, watchOS
2	CIFAR10	60,000	AlexNet-tflite	TensorFlow Lite	Android, Linux-based Systems
3	CIFAR10	60,000	VGG16-coreml	Core ML	iOS, watchOS
4	CIFAR10	60,000	VGG16-tflite	TensorFlow Lite	Android, Linux-based Systems
5	Fashion	70,000	LeNet1-coreml	Core ML	iOS, watchOS
6	Fashion	70,000	LeNet1-tflite	TensorFlow Lite	Android, Linux-based Systems
7	Fashion	70,000	LeNet5-coreml	Core ML	iOS, watchOS
8	Fashion	70,000	LeNet5-tflite	TensorFlow Lite	Android, Linux-based Systems
9	Plant	52,803	NIN-coreml	Core ML	iOS, watchOS
10	Plant	52,803	NIN-tflite	TensorFlow Lite	Android, Linux-based Systems
11	Plant	52,803	VGG19-coreml	Core ML	iOS, watchOS
12	Plant	52,803	VGG19-tflite	TensorFlow Lite	Android, Linux-based Systems
13	CIFAR100	60,000	ReseNet152-coreml	Core ML	iOS, watchOS
14	CIFAR100	60,000	ReseNet152-tflite	TensorFlow Lite	Android, Linux-based Systems
15	CIFAR100	60,000	DenseNet201-coreml	Core ML	iOS, watchOS
16	CIFAR100	60,000	DenseNet201-tflite	TensorFlow Lite	Android, Linux-based Systems
17	News	21,107	LSTM-coreml	Core ML	iOS, watchOS
18	News	21,107	LSTM-tflite	TensorFlow Lite	Android, Linux-based Systems
19	News	21,107	GRU-coreml	Core ML	iOS, watchOS
20	News	21,107	GRU-tflite	TensorFlow Lite	Android, Linux-based Systems

effectiveness of PriCod?
In this research question, we investigate the impact of various feature types on the performance of PriCod. This investigation enables us to identify the features that have a more significant influence on PriCod. Additionally, conducting a thorough analysis of feature contributions can enhance our comprehension of the underlying mechanisms and operational principles of PriCod.

- **RQ6: To what extent can uncertainty-based metrics contribute to improving the effectiveness of PriCod?**
  In the test prioritization process within PriCod, we generate embedding features for each test to indirectly reveal the proximity between the test and the decision boundary. A prior study [9] indicated that uncertainty-based metrics can also reflect the proximity. Therefore, in this research question, we investigate whether incorporating uncertainty-based metrics can enhance the effectiveness of PriCod. To be more specific, we employ several uncertainty metrics (such as DeepGini [6] and Margin [88]) to produce uncertainty features for each test and integrate them into the original PriCod for test prioritization.

5.4.2 Models and Datasets

In our study, we have utilized a total of 182 subjects to assess the performance of PriCod and the compared approaches [6, 101]. Table 5.2 provides essential particulars regarding these subjects, encompassing the dataset-model associations, dataset sizes, tools used for DNN compression, and supported mobile platforms. Among the 182 subjects under investigation in this study, 20 subjects are constructed based on natural datasets, 126 subjects are built on noisy datasets, and 36 subjects are established on adversarial datasets.

5.4.2.1 Datasets

In our study, we evaluate the performance of PriCod using five distinct datasets: CIFAR10 [194], Fashion [187], Plant [195], CIFAR100 [196], and News [197]. The selection of these specific datasets is grounded in their widespread adoption within the domain of DNN testing. Notably, CIFAR10 and Fashion are two of the most

widely employed datasets for DNN evaluation. The Plant dataset stands out as a renowned dataset for AI applications focused on detecting plant diseases. Moreover, compressed models tailored to this context have already been implemented. Therefore, investigating this dataset becomes particularly valuable for the study of compressed models.

- **CIFAR10** [194] The CIFAR10 dataset serves as a frequently utilized collection of images for training and assessing DNN models. Comprising a total of 60,000 images, each measuring 32x32 pixels and in color, the dataset is categorized into ten distinct classes, containing 6,000 images per class. These categories include airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.
- **Fashion** [187] The Fashion dataset comprises fashion product images from 10 categories, such as T-shirts, trousers, dresses, etc. Each category consists of 7000 images, making a total of 70,000 images. All images in the dataset are grayscale with dimensions of 28x28 pixels.
- **Plant** [195] The Plant dataset covers a diverse range of crops and various disease types, including images depicting healthy plant leaves. Specifically, the Plant dataset comprises 52,803 plant leaf images organized into 38 distinct category labels. Each label corresponds to a combination of a specific crop and a disease. Examples of these classes encompass Healthy Corn Leaf, Potato Late Blight, and Rose Black Spot.
- **CIFAR100** [196] The CIFAR100 dataset is a widely used benchmark in computer vision, consisting of 60,000 32x32 color images categorized into 100 distinct classes (such as clouds, cups, and forests). CIFAR100 is a valuable resource for training and evaluating DNN models, particularly designed for image classification tasks.
- **News** [197] The News dataset is an English-language dataset that comprises an annotated corpus of finance-related tweets. This dataset is utilized for the classification of finance-related tweets based on their topics. It consists of 21,107 samples categorized into 20 classes. Examples of these categories include financials, currencies, and opinion.

5.4.2.2 Compressed DNN models

Our study encompasses a set of 20 compressed deep neural network (DNN) models. Our approach for model compression primarily focuses on model quantization, a prevalent method in the field of model compression. The reason for selecting quantization as our compression method is that, in the industry, quantization is considered one of the most commonly adopted techniques for deploying models to mobile devices [70, 198]. In the process of generating compressed models, we primarily employed two quantization techniques to compress DNN models. Below, we provide details regarding the two techniques used to compress DNN models:

- **Tensorflow Lite** [199] We utilized TensorFlow Lite (TFLite) as one of the compression techniques to compress DNN models. TFLite, a component of the TensorFlow deep learning framework developed and maintained by Google, offers interfaces for converting TensorFlow models into lightweight counterparts. This facilitates deployment on various low-computing devices, including Android mobile phones. In our experiments, we chose 8-bit quantization for model compression. This involves quantizing the weights and activation values in the model to 8-bit numbers, thereby reducing the model's size and enhancing deployment efficiency

on resource-constrained devices.

- **CoreML** [200] In our experiments, we utilized another pivotal technique for model compression, namely CoreML. Developed by Apple, CoreML is a framework designed to transform models into the Mlmodel format, customized for iOS platforms. Similar to parameter selection in TFLite, we implemented 8-bit quantization. This process involves quantizing the model's weights and activation values into 8-bit numbers.

In the following, we present detailed information on all the compressed DNN models employed to evaluate PriCod.

- **AlexNet-coreml and AlexNet-tflite** AlexNet [201] is a deep convolutional neural network designed for image classification tasks. It comprises multiple convolutional layers, pooling layers, and fully connected layers, all using the Rectified Linear Unit (ReLU) activation function. ***AlexNet-coreml*** is a compressed version of the AlexNet model converted into the CoreML format, suitable for inference and applications on Apple devices. ***AlexNet-tflite*** is a compressed version of the AlexNet model converted into the TensorFlow Lite (TFLite) format, designed for inference and applications on Android and embedded devices.
- **VGG16-coreml and VGG16-tflite** VGG16 [202] is a deep convolutional neural network with 16 convolutional and fully connected layers, primarily used for image classification. Its notable features include fixed 3x3 convolutional kernel size and a large number of layers, making it suitable for training on large-scale image datasets. ***VGG16-coreml*** is a compressed version of the VGG16 model converted into the CoreML format intended for image processing tasks on Apple devices. ***VGG16-tflite*** is a compressed version of the VGG16 model converted into the TFLite format, designed for image processing tasks on Android and embedded devices.
- **LeNet1-coreml and LeNet1-tflite** LeNet-1 [203] is an early convolutional neural network. It consists of convolutional layers, pooling layers, and fully connected layers, suitable for small-scale image classification tasks. ***LeNet1-coreml*** is a compressed version of the LeNet-1 model converted into the CoreML format intended for image processing and recognition tasks on Apple devices. ***LeNet1-tflite*** is a compressed version of the LeNet-1 model converted into the TFLite format.
- **LeNet5-coreml and LeNet5-tflite** LeNet-5 [203] is another early convolutional neural network. It includes convolutional layers, pooling layers, and fully connected layers and has more layers and parameters compared to LeNet-1. ***LeNet5-coreml*** is a compressed version of the LeNet-5 model converted into the CoreML format. ***LeNet5-tflite*** is a compressed version of the LeNet-5 model converted into the TFLite format.
- **NIN-coreml and NIN-tflite** NIN [204] is an innovative convolutional neural network architecture that introduces the concept of "network in network" to enhance model expressiveness. It uses 1x1 convolutional layers to extract local features. ***NIN-coreml*** is a compressed version of the NIN model converted into the CoreML format, suitable for image processing tasks on Apple devices. ***NIN-tflite*** is a compressed version of the NIN model converted into the TFLite format.
- **VGG19-coreml and VGG19-tflite** VGG19 [202] is an extended version of VGG16 with more convolutional and fully connected layers, suitable for complex

image classification tasks. ***VGG19-coreml*** is a compressed version of the VGG19 model converted into the CoreML format. ***VGG19-tflite*** is a compressed version of the VGG19 model converted into the TFLite format.

- **ReseNet152-coreml and ReseNet152-tflite** ResNet152 [87] is a deep convolutional neural network (CNN) employing the Residual Network architecture, with 152 layers and 60.4 million parameters. ***ResNet152-coreml*** represents a compressed version of the ResNet152 model, converted into the CoreML format. Similarly, ***ResNet152-tflite*** is a compressed variant of the ResNet152 model, converted into the TFLite format.
- **DenseNet201-coreml and DenseNet201-tflite** DenseNet201 [205] is a convolutional neural network consisting of 201 layers and a total of 20.2 million parameters. ***DenseNet201-coreml*** denotes a compressed version of the DenseNet201 model, converted into the CoreML format. ***DenseNet201-tflite*** is another compressed version of the DenseNet201 model, converted into the TFLite format.
- **LSTM-coreml and LSTM-tflite** [206] LSTM (Long Short-Term Memory) is a type of recurrent neural network known for its ability to capture long-term dependencies in sequential data. ***LSTM-coreml*** refers to a compressed version of the LSTM model, converted into the CoreML format. ***LSTM-tflite*** is another compressed version of the LSTM model, converted into the TFLite format.
- **GRU-coreml and GRU-tflite** [207] GRU (Gated Recurrent Unit) is a recurrent neural network architecture widely employed for processing sequential data. Compared to LSTM, GRU is characterized by fewer gates and parameters, making it faster. ***GRU-coreml*** denotes the compressed version of the GRU model, converted into the CoreML format. ***GRU-tflite*** is another compressed version of the GRU model, converted into the TFLite format.

5.4.3 Noise Generation Techniques

In our experiments for Research Question 2 (RQ2), we utilized 13 noise techniques sourced from top-level conferences [93, 94, 95, 96]. These diverse selections of noise generation techniques were aimed at evaluating the effectiveness of PriCod in a broader range of noisy scenarios. We provide a detailed explanation of each noise-generation technique below.

- **Channel Shift Range (CSR)** The CSR technique engenders a transformative alteration in the image's overall color palette by perturbing the values of its color channels.
- **Feature-wise Std Normalization (FSN)** FSN operates by normalizing each input sample with respect to its standard deviation. The underlying motivation is to decentralize the dataset.
- **Height Shift (HS)** HS effectuates vertical displacements of an image, essentially shifting it upwards or downwards within the image canvas. This spatial modification introduces variations in the vertical positioning of objects.
- **Horizontal Flip (HF)** The HF technique orchestrates horizontal mirroring of the input image. By introducing random horizontal flips during augmentation, diverse perspectives of objects are captured.
- **Vertical Flip (VF)** VF introduces a vertical inversion of the image, essentially flipping it along the horizontal axis.
- **Rotation (RO)** RO introduces controlled rotations to the input samples, adhering to a designated angle range.

- **Shear Range (SR)** SR engenders a shear transformation, preserving one coordinate while linearly displacing the other.
- **Width Shift (WS)** WS pertains specifically to horizontal translations, thereby repositioning the image horizontally within the canvas.
- **Zca Whitening (ZCA)** ZCA performs dimensionality reduction on the input images, effectively reducing redundancy while retaining essential features.
- **Zoom (ZOO)** ZOO introduces alterations in the image scale by magnifying or contracting it along its length or width.
- **Contrast (CON)** CON quantifies the disparity between the brightest and darkest regions of an image. This augmentation manipulates image contrast, diversifying the dataset concerning luminance and accentuating differences between light and dark areas.
- **Noise Gasuss (GAS)** GAS simulates signal noise by following a Gaussian distribution. By adding this form of noise to the input images, the dataset's resilience to stochastic variations is bolstered.
- **Salt & Pepper (SP)** SP augments the dataset by introducing either white or black pixels to the image, imitating the effects of salt and pepper noise. This introduces localized distortions.

5.4.4 Adversarial Techniques

In the context of RQ3 in our study, we employed four distinct adversarial techniques to generate adversarial samples for assessing the effectiveness of PriCod. These techniques include the Fast Gradient Method, Adversarial Patch, Basic Iterative Method and Projected Gradient Descent. We elaborate on the operational principles of each adversarial technique in the following explanations.

- **Fast Gradient Method (FGM)** [208] FGM is an extension of the Fast Gradient Sign Method (FGSM), which was the pioneering gradient-based white-box attack algorithm utilizing deep neural network gradients to craft adversarial examples. FGM enhances the original FGSM attack by incorporating other norms for perturbation generation.
- **Adversarial Patch (Patch)** [209] AP generates adversarial examples by creating attack patches designed to replace specific portions of the original images. Importantly, this technique does not necessitate attackers to possess knowledge about the original dataset.
- **Basic Iterative Method (BIM)** [210] BIM is an advancement of FGSM involving multiple iterations of small perturbations. After each iteration, the pixel values of the obtained result are clipped to ensure that the outcome remains within the vicinity of the original image.
- **Projected Gradient Descent (PGD)** [159] PGD attack is an iterative technique. In contrast to FGSM, which involves a single iteration and a significant perturbation, PGD incorporates multiple iterations with small perturbations. During each iteration, the perturbation is constrained within predefined boundaries.

5.4.5 Compared Approaches

To demonstrate the effectiveness of PriCod, we employed seven test prioritization approaches along with a baseline method. These seven methods are DeepGini, Prediction-Confidence Score (PCS), Vanilla Softmax, Entropy, Margin, Least Confidence (LC) and ATS. We chose these methods for the following reasons: 1) These

approaches can be tailored to prioritize tests for compressed Deep Neural Networks; 2) These approaches have previously shown effectiveness for DNNs; 3) These approaches offer open-source implementations. It is crucial to emphasize that all the test prioritization methods used for comparison were initially designed for non-compressed models in their respective research papers. However, despite being designed for uncompressed models, these methods can be directly applied to compressed DNN models. This is a crucial factor why we selected them to compare with PriCod.

- **DeepGini** [6] DeepGini performs test prioritization by calculating the model confidence towards each test case. The metric used to measure the confidence score is the Gini coefficient, which is calculated using the Formula 6.19 provided below. A higher Gini coefficient for a test indicates that the test is more likely to be misclassified. Therefore, it should be prioritized towards the front of the test set.

$$Gini(t) = 1 - \sum_{i=1}^{N} (p_i(t))^2 \tag{5.1}$$

  where $Gini(t)$ represents the Gini score of the test t. $p_i(t)$ denotes the probability that the test input t is predicted to belong to label i. N represents the total number of classes to which the input can be classified.
- **Prediction-Confidence Score (PCS)** [9] For each test input in the specified test set, PCS calculates the difference between the probability of the model's most confident prediction for that test and the probability of the second most confident prediction. A smaller difference indicates that the model is less confident in its prediction for that particular test input, and this input will be prioritized higher. The formula for this calculation is provided in Formula 6.20.

$$PCS(t) = p^1(t) - p^2(t) \tag{5.2}$$

  where $p^1(t)$ denotes the probability of the model's most confident prediction for that test, and $p^2(t)$ represents the probability of the model's second most confident prediction for the same test.
- **Vanilla Softmax** [9] Vanilla Softmax prioritizes tests by calculating the difference between the maximum activation probability in the output softmax layer and the ideal value of 1 for each test input. Test inputs exhibiting larger disparities are perceived as more likely to be misclassified by the model. The computation of Vanilla Softmax is demonstrated in Formula 6.21.

$$\mathrm{V}(t) = 1 - \max_{i=1}^{N} p_i(t) \tag{5.3}$$

  where $\max_{i=1}^{N} p_i(t)$ represents the maximum activation probability in the output softmax layer for the test input t. Here, N denotes the total number of prediction classes. $p_i(t)$ signifies the probability that the model classify the test t into class i.
- **Entropy** [9] Entropy prioritizes tests by computing the entropy of the softmax likelihood for each test instance. Higher entropy values imply higher uncertainty in the model's predictions for those inputs. Therefore, test inputs with greater entropy are interpreted as more prone to being misclassified by the model and will be assigned higher priority.
- **Margin** [88] Margin prioritizes tests by evaluating the difference between the model's most confident prediction and the second most confident prediction for

each test. For a given test, if its margin score is large, the test is considered more likely to be misclassified. The margin score is calculated by Formula 5.4.

$$M(t) = p_k(t) - p_j(t) \tag{5.4}$$

where $M(x)$ denotes the margin score. $p_k(t)$ represents the model's most confident prediction probability for the test instance t. $p_j(t)$ represents the second most confident prediction probability.

- **Least Confidence (LC)** [88] Least Confidence regards test inputs for which the model exhibits the least confidence as more likely to be misclassified. The least confidence score is calculated using Formula 5.5. For a given test, a higher least confidence score indicates that the model is less confident about the prediction for that particular test. Therefore, this test is considered more likely to be predicted incorrectly.

$$L(t) = 1 - \max_{i=1:n} p_i(t) \tag{5.5}$$

where $L(t)$ represents the least confidence score. $p_i(t)$ denotes the probability that the test input t is predicted to be label i via a model M. $\max_{i=1:n} p_i(t)$ corresponds to the model's most confident prediction probability for the test instance t.

- **ATS** [211] ATS (Adaptive Test Selection) is an adaptive method for test selection that utilizes variations in model outputs to assess the behavioral diversity of DNN test data. Its objective is to select a diverse subset of tests from a massive unlabeled dataset. In empirical evaluations [211], ATS has demonstrated superior performance compared to all the evaluated coverage-based test selection methods, showing significant improvements in both fault detection and model improvement capabilities.
- **Random selection** [102] Random selection serves as the baseline in our study. This approach involves randomly determining the order in which test inputs are executed. This implies that the arrangement of test inputs is established entirely at random, without any predefined patterns or logical sequences.

In addition to the aforementioned test prioritization methods, there are several classic approaches in the literature for prioritizing tests for DNNs, including PRIMA [10] and coverage-based test prioritization methods [8]. However, due to the characteristics of compressed models, these methods are challenging to directly apply. We explain the specific reasons below:

- **PRIMA** PRIMA is not suitable for compressed DNN models primarily because some of its model mutation rules cannot be adapted to the structure of the compressed DNN model. For example, one of PRIMA's model mutation operations is Neuron Activation Inverse, which reverses the activation state of a neuron by changing the sign of the neuron output before passing it to the activation function. However, previous research [19] has pointed out that the structure of compressed models does not support such model mutation operations.
- **Coverage-based metrics** Coverage-based test prioritization methods, such as DeepXplore, cannot be applied to compressed DNN models primarily due to the fact that coverage-based approaches typically prioritize test inputs based on their neuron coverage. However, existing study [19] pointed out that, due to the unique structure of compressed models, obtaining neuron coverage is not feasible. Therefore, coverage-based test prioritization methods cannot be adapted to compressed DNN models.

5.4.6 Measurements

Consistent with previous studies [6], we utilized two metrics to evaluate the effectiveness of PriCod: Average Percentage of Fault Detection (APFD) [11] and Percentage of Faults Detected (PFD) [6].

5.4.6.1 Average Percentage of Fault Detection (APFD)

APFD is a widely accepted metric for evaluating test prioritization effectiveness. A higher APFD value signifies greater effectiveness. APFD values are calculated using Formula 6.17.

$$APFD = 1 - \frac{\sum_{i=1}^{k} o_i}{kN} + \frac{1}{2N} \tag{5.6}$$

- N represents the total number of test inputs in the test set.
- k signifies the number of misclassified test inputs.
- o_i refers to the index of the i_{th} misclassified test within the prioritized test set.

Below, we explain from a mathematical perspective why a larger APFD indicates better effectiveness of a test prioritization method: Given that N is a constant, a higher APFD value corresponds to a smaller $\sum_{i=1}^{k} o_i$ (the index sum of misclassified tests in the prioritized list). A smaller $\sum_{i=1}^{k} o_i$ suggests that misclassified tests are positioned closer to the beginning of the prioritized test set, indicating that the test prioritization approach prioritized misclassified tests higher. Such an approach is considered to exhibit a high level of effectiveness.

Consistent with prior research [6], we normalize APFD values to the range [0, 1]. A prioritization approach is considered more effective if its APFD value approaches 1.

5.4.6.2 Percentage of Fault Detected (PFD)

PFD quantifies the ratio of correctly identified misclassified test inputs to the total count of misclassified tests. A higher PFD value indicates greater effectiveness of a test prioritization approach. PFD is calculated as shown in Formula 6.18.

$$PFD = \frac{F_c}{F_t} \tag{5.7}$$

- F_c represents the number of detected misclassified test inputs
- F_t is the total number of misclassified test inputs

In our investigation, we evaluated the PFD values of PriCod across different ratios of prioritized tests. We use **PFD-n** to denote the first n% prioritized test inputs.

5.4.7 Im plementation and Configuration

We implemented PriCod using Python and the PyTorch 2.0.0 framework [103] and TensorFlow 2.3.1 framework. To facilitate comparisons with other approaches, we integrated existing implementations of the compared methods [9, 6] into our experimental pipeline. The compression of DNN models to the CoreML format was performed on macOS Ventura 13.4.1, as CoreML models can only be executed on iOS systems. For the classifier used in the ranking process, we employed LightGBM 3.3.5 with specific parameter settings: the learning rate of 0.1, *n_estimators* of 100, and *min_child_sample* of 20. Furthermore, we leveraged the packages SciPy 1.4.1 and

scikit-learn 1.1.3 for data processing. Below, we present the test accuracy and training accuracy of the ranking model LightGBM on each dataset: **1) CIFAR10** Training Accuracy: 96.82%~97.84%; Test Accuracy: 77.42%~82.24% **2) CIDAR100** Training Accuracy: 96.28%~97.13%; Test Accuracy: 82.15%~85.58% **3) Fashion** Training Accuracy: 97.53%~97.96%; Test Accuracy: 86.31%~86.75% **4) Plant** Training Accuracy: 97.59%~98.12%; Test Accuracy: 85.51%~89.15% **5) News** Training Accuracy: 94.26%~96.13%; Test Accuracy: 77.79%~80.25%. In our experiments, the accuracy of the compressed DNN models used to evaluate PriCod ranged from 70.03% to 77.43%. The accuracy range of their original DNN model was between 70.23% and 78.74%. Other fundamental information about the models can be found in Table 5.2. Our experimental setup involved conducting experiments on NVIDIA Tesla V100 32GB GPUs. We used a MacBook Pro laptop running macOS Ventura 13.4.1 for data analysis, equipped with an Intel Core i9 CPU and 64 GB of RAM. In total, our study encompassed experiments involving 182 subjects, with 20 subjects based on natural inputs, 126 subjects based on noisy inputs, and 36 subjects based on adversarial inputs.

5.5 Results and analysis

5.5.1 RQ1: Performance of PriCod on Natural Test Inputs

Objectives: We evaluate the effectiveness of PriCod on natural test inputs with 20 compressed DNN models. We compare PriCod with a set of existing test prioritization approaches and a baseline method (i.e., random selection). Moreover, we evaluate PriCod on compressed DNN models with different accuracy levels, aiming to better assess the effectiveness of PriCod. Specifically, the experiments are conducted based on the following two sub-questions:

- **RQ-1.1** How does PriCod perform in terms of effectiveness and efficiency when applied to natural test inputs?
- **RQ-1.2** How does PriCod perform on compressed DNN models with different accuracy levels?

Experimental design: We conducted the following experiments to answer the aforementioned sub-questions, respectively.

[**Experiment for RQ-1.1**] We assessed the effectiveness and efficiency of PriCod on natural test inputs through the following experimental steps.

- **Subject Construction** We constructed 20 subjects consisting of compressed DNN models and their corresponding datasets. Specifically, we utilized 20 compressed DNN models along with three datasets. For specific details regarding the models and datasets, please refer to Section 6.4.2. The matching relationships are illustrated in Table 3.1.
- **Selection of Comparative Approaches** Subsequently, we meticulously selected five comparative approaches (i.e., DeepGini, Vanilla SM, PCS, entropy, and random selection) from the existing literature [6, 9]. These approaches can be adapted for prioritizing test inputs for compressed DNN models, with random selection as the baseline.
- **Evaluation of Effectiveness** Within our constructed 20 subjects, we evaluated the effectiveness of PriCod and all comparative methods using two widely adopted metrics: Average Percentage of Fault-Detection (APFD) and Percentage of Fault Detected (PFD) [6]. Detailed explanations of these metrics can be found in

Section 6.4.3.

- **Comparison of Efficiency** Moreover, we compared the efficiency of PriCod and other test prioritization methods by analyzing their time costs.
- **Statistical Analysis** Recognizing the inherent variability within the model training process, we undertook a statistical analysis by executing each experiment ten times. We showcase the average outcomes of these trials.
  More specifically, we employed the paired two-sample t-test [104], a widely utilized statistical method for comparing differences between two related datasets. The fundamental steps involve: 1) selecting two sets of related data, 2) calculating the difference for each corresponding pair of data points, and 3) analyzing these differences to assess whether there is a statistically significant disparity between the two datasets. In the context of the paired two-sample t-test, the significance of the results is determined by the p-value. Typically, a p-value less than 10^{-05} indicates a statistically significant difference between the two datasets [105].

[**Experiment for RQ-1.2**] We evaluated PriCod on compressed DNN models with varying accuracy levels using the following experimental steps. Initially, we trained a group of compressed DNN models at various accuracy levels, specifically 50%, 60%, 70%, 80%, and 90%. We evaluated the effectiveness of PriCod separately for each accuracy level using the APFD metric. Subsequently, we presented and compared PriCod's effectiveness across different accuracy levels through tabular representation.

Table 5.3: Effectiveness comparison among PriCod, DeepGini, VanillaSM, PCS, Entropy, Margin, LC, ATS and random selection in terms of the APFD values on natural test inputs

Data	Model	Approach								
		Random	DeepGini	VanillaSM	PCS	Entropy	Margin	LC	ATS	**PriCod**
CIFAR10	AlexNet-coreml	0.501	0.668	0.666	0.658	0.669	0.658	0.666	0.599	0.721
	AlexNet-tflite	0.502	0.670	0.668	0.660	0.671	0.660	0.668	0.601	0.720
	VGG16-coreml	0.497	0.748	0.747	0.744	0.747	0.744	0.747	0.708	0.781
	VGG16-tflite	0.501	0.750	0.749	0.747	0.748	0.747	0.749	0.707	0.781
CIFAR100	DenseNet201-coreml	0.502	0.737	0.737	0.734	0.735	0.734	0.737	0.712	0.788
	DenseNet201-tflite	0.501	0.753	0.755	0.753	0.747	0.753	0.755	0.703	0.796
	ResNet152-coreml	0.498	0.710	0.711	0.707	0.703	0.707	0.711	0.688	0.765
	ResNet152-tflite	0.496	0.749	0.751	0.746	0.741	0.746	0.751	0.691	0.786
Fashion	LeNet1-coreml	0.502	0.743	0.744	0.742	0.737	0.742	0.744	0.616	0.815
	LeNet1-tflite	0.504	0.743	0.744	0.741	0.737	0.741	0.744	0.615	0.815
	LeNet5-coreml	0.508	0.763	0.763	0.757	0.760	0.757	0.763	0.623	0.826
	LeNet5-tflite	0.496	0.763	0.763	0.757	0.760	0.757	0.763	0.626	0.824
Plant	NIN-coreml	0.501	0.740	0.743	0.743	0.736	0.743	0.743	0.711	0.795
	NIN-tflite	0.501	0.742	0.744	0.744	0.737	0.744	0.744	0.714	0.794
	VGG19-coreml	0.497	0.687	0.685	0.683	0.687	0.683	0.685	0.643	0.779
	VGG19-tflite	0.502	0.688	0.687	0.684	0.689	0.684	0.687	0.645	0.781
News	GRU-coreml	0.491	0.713	0.715	0.713	0.707	0.713	0.715	0.684	0.756
	GRU-tflite	0.505	0.713	0.715	0.712	0.707	0.712	0.715	0.685	0.757
	LSTM-coreml	0.503	0.729	0.731	0.732	0.723	0.732	0.731	0.691	0.771
	LSTM-tflite	0.511	0.729	0.732	0.733	0.723	0.733	0.732	0.692	0.770

Results: The experimental results for RQ-1.1 are depicted in Table 5.3, Table 5.4, Table 5.5, Table 5.6, and Table 5.7. Among these, the first two tables compare PriCod and other test prioritization methods based on the APFD metric using natural test inputs. Table 5.5 presents a comparison using the PFD metric. Table 5.6 presents detailed results from the statistical analysis in terms of PFD. Table 5.7 illustrates the comparison in terms of efficiency.

PriCod consistently outperforms all the compared approaches (i.e., DeepGini, Vanilla SM, PCS, Entropy, and Random) in terms of effectiveness. Table 5.3 showcases the effectiveness comparison of PriCod and all comparative

Table 5.4: Average improvement of PriCod over the compared approaches in terms of the APFD values on natural test inputs

Approach	# Best cases	Average APFD	Improvement(%)
Random	0	0.501	55.89
DeepGini	0	0.726	7.58
VanillaSM	0	0.727	7.43
PCS	0	0.724	7.87
Entropy	0	0.723	8.02
Margin	0	0.724	7.87
LC	0	0.727	7.43
ATS	0	0.668	16.92
PriCod	20	0.781	-

methods across different subjects, measured using the APFD metric. Notably, we highlighted the approach with the highest effectiveness in gray for each case. From Table 5.3, we see that PriCod performs better than all the comparative methods across all subjects. The range of PriCod's APFD values spans from 0.720 to 0.826, while the APFD values for the comparative methods fall within the range of 0.491 to 0.763. Table 5.4 delves deeper into the effectiveness comparison between PriCod and other test prioritization methods. This comparison is approached from three perspectives: the number of cases in which each approach performs the best, the average APFD of each method, and the average improvement in APFD of PriCod over the compared test prioritization methods. Notably, the average APFD value achieved by PriCod is 0.781, with an average improvement ranging from 7.43% to 55.89% over the compared test prioritization approaches. Table 5.5 displays the comparative results between PriCod and existing test prioritization methods utilizing the metric PFD. We see that PriCod showcases a higher level of effectiveness across different prioritized test ratios, surpassing all the compared techniques. The aforementioned observations strongly demonstrate that PriCod outperforms all the compared approaches in terms of both APFD and PFD.

As mentioned in the experimental design, we conducted a statistical analysis to evaluate the stability of our findings. This involved repeating all experiments ten times. All results presented are the average values obtained from these ten repetitions. Furthermore, we identified that the p-value is less than 10^{-05}, underscoring the consistent superiority of PriCod over the compared methods in test prioritization.

Moreover, Table 5.6 presents detailed results from the statistical analysis in terms of PFD. We see that all the p-values between PriCod and the compared approaches consistently fall below 10^{-05}, indicating that PriCod statistically outperforms all the compared methods in terms of PFD. For example, the p-value for the difference in experimental results between PriCod and DeepGini is 3.22×10^{-06}. The p-value between PriCod and PCS is 6.69×10^{-06}.

The efficiency of PriCod falls within an acceptable range. Table 5.7 presents a comparison of the efficiency between PriCod and the compared methods. We observe that PriCod's total execution time is less than 9 min 20s, which can be divided into three main components: Feature generation, training, and prediction. Among these, feature generation takes 9 minutes, training requires 18 seconds, and prediction is completed in less than 1 second. The final prediction time for the compared methods is less than 1 second. While PriCod is not as efficient as the confidence-based test prioritization approaches, its efficiency falls within an acceptable range.

Table 5.5: Average comparison results among PriCod and the compared approaches in terms of PFD on natural test inputs

Data	Approach	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60	PFD-70
CIFAR10	Random	0.105	0.201	0.302	0.402	0.498	0.599	0.699
	DeepGini	0.230	0.418	0.577	0.713	0.819	0.896	0.945
	VanillaSM	0.230	0.414	0.575	0.710	0.818	0.895	0.945
	PCS	0.213	0.403	0.568	0.705	0.814	0.893	0.943
	Entropy	0.226	0.415	0.575	0.714	0.821	0.898	0.947
	Margin	0.213	0.403	0.568	0.705	0.814	0.893	0.943
	LC	0.230	0.414	0.575	0.710	0.818	0.895	0.945
	ATS	0.206	0.367	0.503	0.612	0.711	0.807	0.901
	PriCod	0.273	0.489	0.661	0.786	0.875	0.935	0.971
CIFAR100	Random	0.102	0.199	0.299	0.402	0.501	0.601	0.699
	DeepGini	0.249	0.459	0.632	0.769	0.869	0.935	0.972
	VanillaSM	0.253	0.462	0.635	0.771	0.870	0.935	0.972
	PCS	0.241	0.452	0.630	0.769	0.869	0.934	0.972
	Entropy	0.245	0.449	0.619	0.756	0.859	0.929	0.971
	Margin	0.241	0.452	0.630	0.769	0.869	0.934	0.972
	LC	0.253	0.462	0.635	0.771	0.870	0.935	0.972
	ATS	0.221	0.416	0.581	0.702	0.784	0.854	0.913
	PriCod	0.283	0.521	0.711	0.847	0.931	0.973	0.991
Fashion	Random	0.102	0.197	0.299	0.397	0.501	0.599	0.699
	DeepGini	0.258	0.479	0.665	0.805	0.893	0.944	0.974
	VanillaSM	0.259	0.483	0.666	0.804	0.893	0.944	0.973
	PCS	0.248	0.468	0.659	0.801	0.891	0.942	0.974
	Entropy	0.253	0.472	0.654	0.791	0.890	0.942	0.974
	Margin	0.248	0.468	0.659	0.801	0.891	0.942	0.974
	LC	0.259	0.483	0.666	0.804	0.893	0.944	0.973
	ATS	0.219	0.344	0.425	0.521	0.612	0.713	0.864
	PriCod	0.356	0.626	0.812	0.919	0.968	0.988	0.996
Plant	Random	0.101	0.199	0.298	0.399	0.502	0.603	0.703
	DeepGini	0.219	0.412	0.575	0.718	0.834	0.911	0.963
	VanillaSM	0.221	0.409	0.578	0.721	0.834	0.912	0.963
	PCS	0.214	0.406	0.577	0.722	0.834	0.913	0.964
	Entropy	0.217	0.410	0.576	0.713	0.828	0.909	0.961
	Margin	0.214	0.406	0.577	0.722	0.834	0.913	0.964
	LC	0.221	0.409	0.578	0.721	0.834	0.912	0.963
	ATS	0.201	0.362	0.523	0.657	0.763	0.851	0.912
	PriCod	0.281	0.531	0.735	0.876	0.949	0.983	0.994
News	Random	0.095	0.194	0.291	0.393	0.493	0.595	0.695
	DeepGini	0.239	0.446	0.618	0.740	0.833	0.898	0.946
	VanillaSM	0.249	0.451	0.623	0.739	0.833	0.897	0.946
	PCS	0.243	0.453	0.623	0.738	0.834	0.897	0.945
	Entropy	0.233	0.429	0.598	0.732	0.830	0.895	0.947
	Margin	0.243	0.453	0.623	0.738	0.834	0.897	0.945
	LC	0.249	0.451	0.623	0.739	0.833	0.897	0.946
	ATS	0.212	0.415	0.531	0.672	0.725	0.801	0.887
	PriCod	0.281	0.501	0.666	0.791	0.864	0.926	0.965

Answer to RQ1.1: *PriCod consistently outperforms all the compared approaches (i.e., DeepGini, Vanilla SM, PCS, Entropy, and Random), with an average improvement of 8.28% to 56.69% in terms of APFD. Moreover, the efficiency of PriCod falls within an acceptable range.*

The experimental results for RQ-1.2 are displayed in Table 5.8. In each case, we highlighted the approach with the highest effectiveness in gray. In Table 5.8, we see that, across compressed DNN models at different accuracy levels, PriCod consistently exhibits the highest effectiveness, as measured by APFD. Specifically, PriCod achieves an average APFD of 0.793 across all accuracy levels, while the average APFD for the comparison methods ranges from 0.498 to 0.732. These experimental findings indicate that, across models with different accuracy levels, PriCod's effectiveness surpasses all compared testing prioritization methods.

The methods used for comparison, including DeepGini, VanillaSM, PCS, Entropy, Margin, LC, and ATS, are influenced by the accuracy of the compressed models.

Table 5.6: Statistical analysis on natural test inputs (in terms of p-value on PFD)

	Approach							
	Random	DeepGini	VanillaSM	PCS	Entropy	Margin	LC	ATS
Pricod	3.88×10^{-14}	3.22×10^{-6}	1.21×10^{-7}	6.69×10^{-6}	5.12×10^{-6}	6.69×10^{-6}	1.21×10^{-7}	2.53×10^{-9}

Table 5.7: Time cost of PriCod and the compared test prioritization approaches

Time cost	Approach								
	PriCod	Random	DeepGini	VanillaSM	PCS	Entropy	Margin	LC	ATS
Feature generation	9 min	-	-	-	-	-	-	-	-
Ranking model training	18 s	-	-	-	-	-	-	-	-
Prediction	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s	>10 min

For instance, DeepGini exhibits an APFD of 0.862 in compressed models with 90% accuracy, while it decreases to 0.604 in compressed models with 50% accuracy. Similarly, PCS has an APFD of 0.861 in compressed models with 90% accuracy, and it decreases to 0.595 in compressed models with 50% accuracy. In contrast, PriCod's performance is relatively less affected by the accuracy of the compressed models compared to these methods. For instance, in compressed models with 90% accuracy, PriCod has an APFD of 0.864. In compressed models with 50% accuracy, its APFD is 0.713.

Answer to RQ1.2: *Across compressed DNN models with varying accuracy levels, PriCod consistently performs better than all the compared test prioritization methods.*

Table 5.8: Average Effectiveness comparison among PriCod, DeepGini, VanillaSM, PCS, Entropy, Margin, LC, ATS, and random selection in terms of the APFD values on different accuracy compressed models

Approach	**Accuracy**					**Average APFD**
	50%	60%	70%	80%	90%	
Random	0.501	0.499	0.497	0.501	0.493	0.498
DeepGini	0.604	0.658	0.739	0.784	0.862	0.729
VanillaSM	0.604	0.664	0.742	0.786	0.863	0.732
PCS	0.595	0.656	0.736	0.782	0.861	0.726
Entropy	0.603	0.648	0.732	0.780	0.859	0.724
Margin	0.595	0.656	0.736	0.782	0.861	0.726
LC	0.604	0.664	0.742	0.786	0.863	0.732
ATS	0.586	0.623	0.661	0.735	0.806	0.682
PriCod	0.713	0.751	0.801	0.839	0.864	0.793

5.5.2 RQ2: Effectiveness on Noisy Test Inputs

Objectives: When deployed on mobile devices, compressed DNN models can encounter noisy data due to various factors. These factors encompass user behaviors, such as capturing photos from various angles. All these elements have the potential to introduce noise into the images. As a result, it becomes crucial to evaluate the effectiveness of PriCod using noisy datasets. To accomplish this, we utilize 13 noise generation techniques [93, 94, 95, 96] to construct datasets with inherent noise for the purpose of assessment.

Experimental design: We introduce noise to the original datasets using 13 noise generation techniques collected from existing literature. Specifically, given an original test set, we extract 30% of the tests and transform them into noisy data using a noise

technique, while the remaining 70% are left unchanged. The comparative methods we employed in this research question, along with the evaluation metrics, remained consistent with RQ1. The comparative methods were DeepGini, Vanilla SM, PCS, entropy, and random selection. Moreover, we utilized the APFD and PFD metrics (cf. Section 6.4.3) to evaluate the effectiveness of PriCod.

Table 5.9: Average Effectiveness comparison among PriCod, DeepGini, VanillaSM, PCS, Entropy, Margin, LC, ATS, and random selection in terms of the APFD values on noisy test inputs

Noisy Techniques	**Approach**								
	Random	DeepGini	VanillaSM	PCS	Entropy	Margin	LC	ATS	**PriCod**
CSR	0.501	0.724	0.724	0.720	0.721	0.720	0.724	0.645	0.775
FSN	0.499	0.729	0.729	0.724	0.726	0.724	0.729	0.650	0.779
HS	0.499	0.708	0.708	0.703	0.706	0.703	0.708	0.641	0.767
HF	0.499	0.693	0.692	0.687	0.692	0.687	0.692	0.637	0.762
VF	0.500	0.675	0.673	0.667	0.675	0.667	0.673	0.609	0.737
RO	0.499	0.689	0.688	0.681	0.690	0.681	0.688	0.626	0.753
SR	0.499	0.729	0.729	0.724	0.726	0.724	0.729	0.649	0.779
WS	0.497	0.711	0.710	0.704	0.710	0.704	0.710	0.634	0.771
ZCA	0.500	0.729	0.729	0.724	0.726	0.724	0.729	0.651	0.778
ZOO	0.501	0.684	0.681	0.674	0.686	0.674	0.681	0.608	0.748
CON	0.500	0.704	0.703	0.698	0.703	0.698	0.703	0.672	0.746
GAS	0.499	0.699	0.696	0.689	0.700	0.689	0.696	0.614	0.759
SP	0.500	0.711	0.711	0.706	0.709	0.706	0.711	0.638	0.771

Table 5.10: Average improvement of PriCod over the compared approaches in terms of the APFD values on noisy test inputs

Approach	**# Best cases**	**Average APFD**	**Improvement(%)**
Random	0	0.499	52.91
DeepGini	0	0.707	7.92
VanillaSM	0	0.706	8.07
PCS	0	0.701	8.84
Entropy	0	0.705	8.23
Margin	0	0.701	8.84
LC	0	0.706	8.07
ATS	0	0.636	19.97
PriCod	126	0.763	-

Results: The experimental results for RQ2 are presented in Table 5.9, Table 5.10, Table 5.11, and Table 5.12. When applied to noisy test inputs, PriCod consistently exhibits superior performance compared to all the test prioritization approaches under different noise generation techniques. Specifically, Table 5.9 and Table 5.10 illustrate the effectiveness of PriCod and the compared test prioritization methods based on the APFD metric. We see that PriCod's APFD values range between 0.737 and 0.779, while the compared test prioritization methods range from 0.497 and 0.729. Furthermore, Table 5.10 offers a more comprehensive analysis, showcasing the best cases achieved by each approach, the average APFD value of each method, and PriCod's effectiveness improvement relative to each comparative method. Notably, PriCod demonstrates the highest effectiveness across all cases. The average APFD value achieved by PriCod is 0.763, while that of the compared approaches falls between 0.499 and 0.707. The improvement achieved by PriCod over the comparative methods varies from 7.92% to 52.91%.

Table 5.11 and Table 5.12 present a comprehensive comparative analysis of PriCod and the compared approaches regarding the PFD metric. In Table 5.11, we exhibit experimental results under seven noises, while the results for other noise scenarios

Table 5.11: Effectiveness comparison results among PriCod and the compared approaches in terms of PFD on noisy test inputs

Noisy Techniques	Approach	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60	PFD-70
CSR	Random	0.100	0.199	0.301	0.403	0.502	0.601	0.702
	DeepGini	0.236	0.436	0.604	0.743	0.846	0.915	0.958
	Entropy	0.232	0.428	0.599	0.736	0.843	0.914	0.959
	PCS	0.226	0.426	0.597	0.737	0.843	0.913	0.957
	VanillaSM	0.238	0.437	0.604	0.742	0.845	0.915	0.958
	Margin	0.226	0.426	0.597	0.737	0.843	0.913	0.957
	LC	0.238	0.437	0.604	0.742	0.845	0.915	0.958
	ATS	0.207	0.352	0.458	0.559	0.654	0.755	0.845
	PriCod	0.299	0.539	0.721	0.845	0.918	0.961	0.983
FSN	Random	0.101	0.200	0.299	0.400	0.501	0.603	0.702
	DeepGini	0.242	0.445	0.616	0.754	0.853	0.918	0.960
	Entropy	0.238	0.439	0.609	0.748	0.851	0.917	0.960
	PCS	0.229	0.433	0.609	0.748	0.850	0.917	0.959
	VanillaSM	0.243	0.445	0.616	0.753	0.852	0.918	0.960
	Margin	0.229	0.433	0.609	0.748	0.850	0.917	0.959
	LC	0.243	0.445	0.616	0.753	0.852	0.918	0.960
	ATS	0.212	0.355	0.464	0.566	0.661	0.759	0.850
	PriCod	0.306	0.547	0.730	0.850	0.923	0.963	0.985
HS	Random	0.099	0.199	0.300	0.399	0.498	0.601	0.701
	DeepGini	0.220	0.407	0.571	0.711	0.823	0.902	0.953
	Entropy	0.217	0.401	0.564	0.706	0.818	0.901	0.952
	PCS	0.205	0.394	0.562	0.704	0.817	0.899	0.951
	VanillaSM	0.218	0.407	0.571	0.709	0.822	0.901	0.953
	Margin	0.205	0.394	0.562	0.704	0.817	0.899	0.951
	LC	0.218	0.407	0.571	0.709	0.822	0.901	0.953
	ATS	0.196	0.347	0.458	0.556	0.647	0.744	0.837
	PriCod	0.278	0.503	0.687	0.818	0.904	0.953	0.980
HF	Random	0.099	0.200	0.300	0.401	0.499	0.599	0.699
	DeepGini	0.218	0.403	0.562	0.697	0.799	0.870	0.921
	Entropy	0.216	0.399	0.559	0.694	0.799	0.871	0.923
	PCS	0.204	0.391	0.555	0.689	0.793	0.867	0.918
	VanillaSM	0.217	0.402	0.562	0.695	0.798	0.869	0.921
	Margin	0.204	0.391	0.555	0.689	0.793	0.867	0.918
	LC	0.217	0.402	0.562	0.695	0.798	0.869	0.921
	ATS	0.196	0.344	0.459	0.560	0.652	0.745	0.830
	PriCod	0.287	0.517	0.698	0.829	0.910	0.957	0.982
RO	Random	0.100	0.200	0.299	0.399	0.498	0.597	0.697
	DeepGini	0.204	0.382	0.541	0.679	0.792	0.876	0.934
	Entropy	0.202	0.382	0.543	0.681	0.793	0.877	0.936
	PCS	0.189	0.365	0.525	0.666	0.784	0.872	0.932
	VanillaSM	0.202	0.378	0.536	0.675	0.790	0.875	0.934
	Margin	0.189	0.365	0.525	0.666	0.784	0.872	0.932
	LC	0.202	0.378	0.536	0.675	0.790	0.875	0.934
	ATS	0.180	0.326	0.440	0.534	0.626	0.726	0.826
	PriCod	0.258	0.474	0.655	0.794	0.888	0.946	0.977
SR	Random	0.099	0.201	0.301	0.400	0.501	0.600	0.700
	DeepGini	0.241	0.445	0.615	0.755	0.853	0.918	0.960
	Entropy	0.237	0.439	0.609	0.747	0.851	0.917	0.960
	PCS	0.228	0.433	0.608	0.748	0.849	0.917	0.958
	VanillaSM	0.242	0.445	0.615	0.753	0.852	0.918	0.960
	Margin	0.228	0.433	0.608	0.748	0.849	0.917	0.958
	LC	0.242	0.445	0.615	0.753	0.852	0.918	0.960
	ATS	0.211	0.355	0.464	0.565	0.660	0.758	0.849
	PriCod	0.305	0.547	0.730	0.850	0.922	0.963	0.985
VF	Random	0.099	0.200	0.300	0.401	0.500	0.601	0.701
	DeepGini	0.188	0.358	0.512	0.652	0.771	0.863	0.927
	Entropy	0.186	0.358	0.513	0.654	0.772	0.865	0.929
	PCS	0.175	0.339	0.496	0.639	0.761	0.857	0.924
	VanillaSM	0.187	0.354	0.508	0.648	0.769	0.862	0.927
	Margin	0.175	0.339	0.496	0.639	0.761	0.857	0.924
	LC	0.187	0.354	0.508	0.648	0.769	0.862	0.927
	ATS	0.167	0.306	0.416	0.510	0.604	0.705	0.807
	PriCod	0.233	0.442	0.619	0.764	0.868	0.935	0.972

can be found on our GitHub[2]. In Table 5.11, we see that, across different noisy techniques, PriCod consistently outperforms the compared approaches in terms of PFD. Moreover, Table 5.12 exhibits that PriCod performs the best across varying proportions of prioritized tests. Notably, when prioritizing 50% of the tests, PriCod can identify 90.3% of misclassified tests, while the competing methods can only identify 50.1% to 81.9% misclassified tests. These experimental results demonstrate that PriCod performs better than all the compared test prioritization methods when applied to noisy test inputs.

[2]https://github.com/yinghuali/PriCod/tree/main/tables

Table 5.12: Average effectiveness comparison results among PriCod and the compared approaches in terms of PFD on noisy data

Approach	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60	PFD-70
Random	0.101	0.199	0.301	0.398	0.501	0.602	0.701
DeepGini	0.220	0.410	0.573	0.711	0.819	0.895	0.946
VanillaSM	0.220	0.408	0.571	0.709	0.818	0.895	0.946
Entropy	0.217	0.406	0.570	0.709	0.818	0.896	0.947
PCS	0.207	0.395	0.562	0.702	0.812	0.892	0.944
Margin	0.207	0.395	0.562	0.702	0.812	0.892	0.944
LC	0.220	0.408	0.571	0.709	0.818	0.895	0.946
ATS	0.193	0.338	0.449	0.550	0.644	0.743	0.837
PriCod	0.277	0.506	0.687	0.818	0.903	0.953	0.981

Answer to RQ2: *When applied to noisy test inputs, PriCod continues to outperform all the compared approaches in terms of both APFD (Average Percentage of Fault Detection) and PFD (Percentage of Fault Detection). The improvement achieved by PriCod over the comparative methods varies from 8.46% to 53.80% in terms of APFD.*

5.5.3 RQ3: Effectiveness on Adversarial Test Inputs

Objectives: Besides evaluating the effectiveness of PriCod on natural and noisy test inputs, following the evaluation methodology of previous test prioritization research [10], we also assess its effectiveness on adversarial test inputs.

Experimental design: To generate adversarial test samples, we utilized four distinct adversarial attack techniques: the Fast Gradient Method (FGM) [208], Adversarial Patch (Patch) [209], Basic Iterative Method (BIM) [210], and Projected Gradient Descent (PGD) [159]. This yielded a set of 32 subjects for evaluation. Consistent with our prior research questions, we conducted a comparative analysis between PriCod and four alternative test prioritization approaches, along with a baseline method (random selection). The effectiveness of these methods was measured using the metrics APFD and PFD.

Table 5.13: Average Effectiveness comparison among PriCod, DeepGini, VanillaSM, PCS, Entropy, Margin, LC, ATS and random selection in terms of the APFD values on adversarial test inputs

Adversarial Attack Techniques	Approach								
	Random	DeepGini	VanillaSM	PCS	Entropy	Margin	LC	ATS	**PriCod**
BIM	0.499	0.727	0.727	0.722	0.725	0.722	0.727	0.654	0.773
FGM	0.501	0.729	0.731	0.724	0.727	0.724	0.731	0.658	0.776
Patch	0.499	0.661	0.660	0.656	0.659	0.656	0.660	0.617	0.721
PGD	0.501	0.725	0.725	0.721	0.722	0.721	0.725	0.652	0.771

Table 5.14: Average improvement of PriCod over the compared approaches in terms of the APFD values on adversarial test inputs

Approach	# Best cases	Average APFD	Improvement(%)
Random	0	0.501	51.59
DeepGini	0	0.710	7.18
VanillaSM	0	0.711	7.03
PCS	0	0.705	7.94
Entropy	0	0.708	7.49
Margin	0	0.705	7.94
LC	0	0.711	7.03
ATS	0	0.645	17.98
PriCod	36	0.761	-

Results: The experimental results for RQ3 are presented in Table 5.13, Table 5.14,

Table 5.15: Average effectiveness comparison results among PriCod and the compared approaches on adversarial test inputs in terms of PFD

Approach	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60	PFD-70
Random	0.099	0.199	0.298	0.399	0.501	0.599	0.701
DeepGini	0.224	0.417	0.582	0.719	0.822	0.895	0.944
Entropy	0.220	0.412	0.576	0.713	0.819	0.894	0.944
PCS	0.213	0.406	0.575	0.713	0.818	0.893	0.943
VanillaSM	0.225	0.416	0.582	0.718	0.821	0.894	0.944
Margin	0.213	0.406	0.575	0.713	0.818	0.893	0.943
LC	0.225	0.416	0.582	0.718	0.821	0.894	0.944
ATS	0.201	0.346	0.459	0.561	0.655	0.754	0.846
PriCod	0.282	0.511	0.690	0.816	0.896	0.946	0.975

and Table 5.15. Among these, Table 5.13 and Table 5.14 illustrate the comparative results between PriCod and the compared test prioritization approaches based on the APFD metric on the adversarial test inputs. Table 5.15 showcases the comparative results based on the PFD metric. From Table 5.13, we see that across different attack techniques, PriCod consistently exhibits the highest effectiveness, with the range of its average APFD scores lying between 0.721 and 0.776. In contrast, the average scores of the compared methods range from 0.499 and 0.731. Within Table 5.14, we see that PriCod outperforms all other test prioritization approaches across all 32 cases. PriCod's average APFD score across all subjects is 0.761, while that of the compared methods ranges from 0.501 and 0.711. Furthermore, PriCod achieves an improvement of 7.03% to 51.59% over all the compared methods.

Table 5.15 illustrates the comparison of effectiveness between PriCod and other test prioritization methods based on the PFD metric. Notably, PriCod consistently demonstrates superior effectiveness across varying test prioritization ratios. Notably, when 50% of the tests are prioritized, PriCod can identify 89.6% of misclassified tests. In contrast, the compared methods only managed to identify 50.1% to 82.2% of misclassified tests under the same conditions. Additionally, with a prioritization of 40% of the tests, PriCod can identify 81.6% of misclassified tests, whereas the compared methods only achieve a range of 39.9% to 71.9% for the same metric. These results collectively demonstrate that in terms of the APFD and PFD metrics, PriCod's effectiveness surpasses that of all compared test prioritization methods.

Answer to RQ3: *When applied to adversarial test inputs, PriCod continues to outperform all the compared test prioritization approaches in terms of both APFD and PFD. PriCod achieves an improvement of 8.24%~55.31% over all the compared approaches.*

5.5.4 RQ4: Impact of fusion strategies

Objectives: We conducted an in-depth study of the impact of different feature fusion methods on the effectiveness of PriCod.

Experimental design: To investigate the impact of different feature fusion methods on the effectiveness of PriCod, we designed three variants of PriCod. Each variant employs a distinct feature fusion approach to combine the embedding features and deviation features. These three variants are denoted as PriCod^a, PriCod^m, and PriCod^c, which respectively utilize addition, multiplication, and cross-multiplication techniques for feature fusion. Apart from the method of feature fusion, the rest of these variants remain consistent with the original PriCod. Subsequently, we evaluated their effectiveness along with the original PriCod on natural datasets.

This comparison enabled us to assess the influence of various fusion methods on the effectiveness of PriCod.

Below, we explain why the studied feature fusion strategies are meaningful.

- **Addition-based feature fusion strategy** The addition-based fusion strategy is meaningful due to several reasons: 1) The addition-based fusion strategy preserves the original information of each feature; 2) the addition-based fusion strategy is simple and efficient to implement; 3) the addition-based fusion strategy is suitable for cases where two sets of features are relatively independent, with each contributing independently to the prediction results.
- **Multiplication-based feature fusion strategy** The multiplication-based fusion strategy is meaningful for various reasons: 1) The utilization of the multiplication operation introduces non-linear transformations, enhancing the capability of the ranking model to grasp more feature relationships. 2) Incorporating the multiplication operation aids in mitigating the influence of irrelevant features. When the value of a feature is small, the multiplication diminishes its overall contribution, thereby reducing its impact.
- **Cross-multiplication-based feature fusion strategy** The cross-multiplication-based fusion strategy is meaningful for various reasons: 1) Cross-multiplication-based operations can introduce stronger information interaction between features. This can help the model better understand the relationships between features, thereby enhancing its representational capacity. 2) By considering all possible pairwise interactions, the model operates in a higher-dimensional feature space, capable of revealing more hidden patterns and relationships.

Table 5.16: Effectiveness comparison among PriCod and PriCod Variants in terms of the APFD values on natural test inputs

Data	Model	Approach			
		PriCod^a	PriCod^m	PriCod^c	**PriCod**
CIFAR10	AlexNet-coreml	0.572	0.689	0.717	0.721
	AlexNet-tflite	0.573	0.686	0.717	0.720
	VGG16-coreml	0.611	0.756	0.780	0.781
	VGG16-tflite	0.610	0.756	0.778	0.781
CIFAR100	DenseNet201-coreml	0.654	0.762	0.785	0.788
	DenseNet201-tflite	0.651	0.752	0.782	0.796
	ResNet152-coreml	0.658	0.749	0.764	0.765
	ResNet152-tflite	0.662	0.751	0.776	0.786
Fashion	LeNet1-coreml	0.755	0.783	0.808	0.815
	LeNet1-tflite	0.754	0.771	0.807	0.815
	LeNet5-coreml	0.760	0.794	0.820	0.826
	LeNet5-tflite	0.759	0.783	0.817	0.824
Plant	NIN-coreml	0.671	0.760	0.793	0.795
	NIN-tflite	0.670	0.757	0.792	0.794
	VGG19-coreml	0.652	0.752	0.776	0.779
	VGG19-tflite	0.652	0.753	0.780	0.781
News	GRU-coreml	0.730	0.719	0.736	0.756
	GRU-tflite	0.726	0.717	0.737	0.757
	LSTM-coreml	0.744	0.734	0.754	0.771
	LSTM-tflite	0.744	0.731	0.754	0.770
Average		0.681	0.748	0.774	0.781

Results: The experimental results for RQ4 are presented in Table 5.16. We have shaded the approach with the highest effectiveness in gray for each case. From the table, we see that the effectiveness of PriCod remains consistently highest across

different subjects. Its APFD values range from 0.720 to 0.826. The variant PriCod^a exhibits an APFD range of 0.572 to 0.760. The variant PriCod^m demonstrates an APFD range of 0.686 to 0.794. The variant PriCod^c shows an APFD range of 0.717 to 0.820. We see that the effectiveness of the PriCod^c variant is second only to the original PriCod. In our experiments, PriCod employs a concatenation approach for feature fusion. This indicates that, compared to addition, multiplication, and cross-multiplication, the concatenation method is more suitable for fusing the deviation features and embedding features of compressed DNN models for test prioritization.

Answer to RQ4: *The effectiveness of PriCod surpasses all variants, indicating that among all feature fusion methods, concatenation is more effective in combining the deviation features and embedding features of compressed DNN models for test prioritization.*

5.5.5 RQ5: Feature contribution analysis

Objectives: We delve into understanding the impact of different feature types on the effectiveness of PriCod in test prioritization. Our exploration centers around two key sub-questions presented below:

- **RQ-5.1** To what extent does each type of features contribute to the effectiveness of PriCod?
- **RQ-5.2** How are the feature types distributed among the top-N most influential features towards the effectiveness of PriCod?

Experimental design: We conducted two experiments to address the aforementioned sub-questions.

[**Experiments for RQ-5.1**] Within the initial PriCod framework, we generated two distinct categories of features: deviation features and embedding features. In order to assess the individual impact of each feature type on the effectiveness of PriCod, we conducted a carefully designed ablation study, following established methodologies as outlined in previous work [107]. Specifically, we excluded one feature type at a time while retaining the other. To elaborate, for the assessment of the contribution made by deviation features, we executed PriCod without incorporating deviation features while keeping the embedding features present. Conversely, to gauge the contribution of embedding features, we ran PriCod without embedding features but retained the deviation features. This meticulous ablation study facilitated a quantitative analysis of the influence exerted by each feature type on the overall effectiveness of PriCod.

[**Experiments for RQ-5.2**] To investigate the distribution of different feature types within the top N contributing features, we leveraged the cover metric of the XGBoost algorithm [13]. The specific experimental procedures are detailed below:

❶ **Feature Importance Calculation** Initially, we employed the cover metric to calculate the importance scores of each feature utilized by PriCod in the context of test prioritization.

❷ **Top-N Feature Selection** Subsequently, we identified the N most important features based on the computed scores.

❸ **Categorization Analysis** Through an analysis of the categorization of these selected features, we delved into the extent to which different feature types contribute to the effectiveness of PriCod.

We provide an outline of how XGBoost measures the importance of features as follows. Within the XGBoost algorithm, the cover metric serves as a fundamental tool for

quantifying feature importance. This metric functions by assessing the average coverage of individual instances across the leaf nodes in decision trees. Essentially, the cover metric evaluates how frequently a specific feature is employed to partition data across all trees within the ensemble. The coverage values assigned to each feature across the entirety of trees are then combined, resulting in a cumulative coverage value. To determine the average coverage of each instance by the leaf nodes, the cumulative coverage value is normalized relative to the total number of instances. Consequently, the coverage value attributed to a particular feature plays a decisive role in establishing its significance within the context of the XGBoost model. Notably, features that exhibit higher coverage values are granted increased importance, shaping the XGBoost algorithm's decision-making process. This systematic approach empowers XGBoost to effectively assess the impact of individual features, contributing to accurate predictions and well-informed decisions.

Furthermore, we conducted a more detailed comparison to assess the contributions of different deviation features on the effectiveness of PriCod. Specifically, we performed comparative experiments through the following process.

❶ Under each subject, we calculate the importance value of each type of deviation feature, which reflects its impact on the effectiveness of PriCod within that specific subject.

❷ For each type of deviation feature, we calculated the sum of its importance values across all subjects. For example, in the case of the CLA feature, we summed up all its importance values across all subjects to obtain the final value.

❸ We normalized the final importance values of all deviation features to compare their contributions. We represented the results in the form of a pie chart. In the pie chart, if the proportion of a deviation feature is higher compared to others, it implies that this feature contributes more to the effectiveness of PriCod.

Table 5.17: Ablation study on different features of PriCod: Embedding Features(EB), Deviation Features (DF). 'w/o' means 'without'

Approach	Datasets					Average
	CIFAR10	CIFAR100	Fashion	Plant	News	
PriCod w/o EB	0.745	0.766	0.791	0.762	0.743	0.761
PriCod w/o DF	0.598	0.615	0.778	0.648	0.618	0.652
PriCod	0.751	0.784	0.820	0.787	0.763	0.781

Results: The experimental results for RQ5.1 are presented in Table 5.17. In this table, 'w/o' stands for 'without.' For instance, 'PriCod w/o EB' indicates the execution of PriCod without generating the embedding features. We highlighted the approach with the highest effectiveness in gray for each case.

Each type of features (i.e., deviation features and embedding features) contribute to the effectiveness of PriCod. As indicated by the results in Table 5.17, the unaltered PriCod model achieves the highest average effectiveness. Notably, the removal of any feature type leads to a reduction in PriCod's effectiveness, highlighting that each type of features plays a role in PriCod's effectiveness. For example, on the CIFAR10 dataset, the original PriCod attains an average APFD value of 0.751. Removing embedding features leads to a decrease in PriCod's average APFD to 0.745, while the absence of deviation features results in a more significant decline to 0.598.

Deviation features make the highest average contributions. Moreover, as indicated in Table 5.17, deviation features demonstrate the most substantial

average contributions to the effectiveness of PriCod. Across all datasets, the impact of removing deviation features on PriCod's effectiveness is the most pronounced. On average, across all cases, the exclusion of deviation features leads to a decrease in APFD of 0.129, while the removal of embedding features only results in a decrease of 0.020. To provide specific examples, consider the CIFAR10 dataset: removing deviation features causes a noteworthy reduction in APFD by 0.153, compared to a minor decrease of 0.006 from removing embedding features. On the CIFAR100 dataset, the removal of deviation features results in a decrease in APFD by 0.169, in contrast to a marginal decrease of 0.018 observed when removing embedding features. Likewise, on the Fashion dataset, the absence of deviation features leads to a drop in APFD by 0.042, whereas removing embedding features leads to a decrease of 0.029. Similarly, on the Plant dataset, removing deviation features causes an APFD decrease of 0.139, whereas removing embedding features results in a decrease of 0.025.

Answer to RQ5.1: *Each type of features (i.e., deviation features and embedding features) contribute to the effectiveness of PriCod. Notably, deviation features make the highest average contributions.*

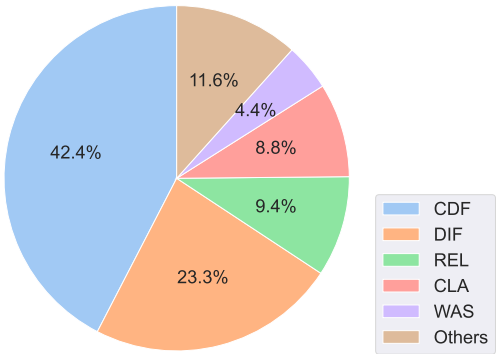


Figure 5.3: Top five contributing features among all deviation features

The results of RQ5.2 are displayed in Table 5.18, where the scores signify the importance levels of individual features. For each pairing of model and dataset, we present the leading N features that contribute significantly. The abbreviations DF and EB denote deviation features and embedding features, respectively. The numerical values appended to the feature abbreviations indicate the corresponding feature indices. For instance, *DF-12* signifies the 12th deviation feature. From Table 5.18, we see that both DF and EB features consistently emerge among the foremost N contributors across diverse subjects. Overall, DF features exhibit a higher overall importance, constituting more than half of an equal share across each subject. Within the CIFAR10 dataset, in the top 10 features with the highest contributions, the proportion of DF features varies from 60% to 70%. In the context of the CIFAR100 dataset, DF features account for 60% to 80% within the top 10 contributing features. Within the News dataset, DF features cover a span of 60% to 70%.

The CDF features have the highest contributions to the effectiveness of PriCod compared to other deviation features. Figure 5.3 illustrates the specific contributions of each deviation feature to the effectiveness of PriCod. In particular, it represents the normalized results of the final importance values for each deviation feature. The detailed calculation process can be found in the experimental design of

Table 5.18: Top-10 most contributing features on the effectiveness of PriCod: Embedding Features(EB), Deviation Features (DF)

Data	Rank	AlexNet-coreml		AlexNet-tflite		VGG16-coreml		VGG16-tflite	
		Feature	Value	Feature	Value	Feature	Value	Feature	Value
CIFAR10	1	DF-0	3022	DF-0	4486	DF-0	2923	DF-4	2876
	2	DF-4	2614	DF-9	4135	DF-4	2768	DF-0	1686
	3	DF-27	2192	DF-4	2896	DF-20	1885	DF-20	1651
	4	DF-13	1655	DF-28	2354	DF-26	1489	DF-26	1294
	5	DF-9	1620	DF-27	2018	DF-5	1459	DF-25	1204
	6	DF-28	1522	EB-170	1822	EB-206	1321	DF-24	1194
	7	EB-121	1460	DF-20	1621	DF-24	1267	DF-22	1164
	8	EB-63	1435	EB-124	1604	EB-175	1252	EB-274	1156
	9	EB-205	1398	DF-23	1557	EB-167	1220	EB-208	1132
	10	DF-25	1387	EB-307	1490	EB-321	1205	EB-59	1127

Data	Rank	DenseNet201-coreml		DenseNet201-tflite		ResNet152-coreml		ResNet152-tflite	
		Feature	Value	Feature	Value	Feature	Value	Feature	Value
CIFAR100	1	DF-86	4563	DF-181	3285	DF-202	2975	DF-91	4874
	2	DF-199	3141	DF-4	3133	DF-163	2974	DF-126	3802
	3	DF-138	3046	DF-184	2825	DF-0	2904	EB-227	3313
	4	DF-159	3012	DF-162	2748	DF-194	2773	DF-4	3107
	5	DF-91	2985	DF-80	2745	DF-122	2678	DF-72	3065
	6	DF-162	2972	DF-141	2734	EB-235	2551	EB-235	3047
	7	DF-4	2849	EB-251	2651	DF-179	2411	EB-233	2905
	8	EB-213	2812	EB-221	2634	DF-4	2351	EB-212	2839
	9	EB-232	2760	DF-125	2597	EB-257	2319	DF-61	2742
	10	DF-0	2749	EB-256	2577	EB-226	2150	DF-5	2633

Data	Rank	LeNet1-coreml		LeNet1-tflite		LeNet5-coreml		LeNet5-tflite	
		Feature	Value	Feature	Value	Feature	Value	Feature	Value
Fashion	1	DF-4	4083	DF-1	4760	DF-4	3685	DF-4	3526
	2	DF-0	3634	DF-4	4110	DF-8	2412	DF-28	1449
	3	DF-25	1943	DF-3	2202	EB-267	1843	EB-275	1417
	4	DF-28	1800	EB-32	1593	EB-258	1705	EB-138	1350
	5	EB-68	1785	EB-204	1585	DF-28	1650	EB-288	1326
	6	DF-5	1779	EB-283	1584	DF-5	1529	EB-55	1324
	7	EB-147	1593	DF-25	1518	EB-240	1326	DF-5	1308
	8	EB-297	1537	EB-262	1432	EB-174	1322	DF-23	1265
	9	EB-206	1531	EB-132	1427	EB-205	1304	EB-319	1262
	10	EB-132	1515	DF-22	1424	EB-133	1277	EB-311	1249

Data	Rank	NIN-coreml		NIN-tflite		VGG19-coreml		VGG19-tflite	
		Feature	Value	Feature	Value	Feature	Value	Feature	Value
Plant	1	DF-0	3335	DF-0	3826	DF-27	1736	DF-4	2834
	2	DF-4	3114	DF-4	2688	DF-4	1729	DF-52	2361
	3	DF-49	1994	DF-53	2632	EB-92	1676	EB-225	1974
	4	DF-76	1947	EB-97	2273	DF-52	1613	EB-256	1867
	5	DF-61	1804	DF-52	2142	DF-50	1582	DF-72	1733
	6	DF-5	1754	EB-88	2112	DF-74	1488	DF-57	1636
	7	EB-87	1748	EB-94	1975	DF-39	1433	DF-9	1587
	8	EB-94	1685	DF-49	1888	EB-247	1428	EB-323	1562
	9	DF-53	1679	DF-55	1784	EB-89	1350	DF-78	1438
	10	EB-148	1670	DF-50	1766	DF-20	1344	EB-264	1421

Data	Rank	GRU-coreml		GRU-tflite		LSTM-coreml		LSTM-tflite	
		Feature	Value	Feature	Value	Feature	Value	Feature	Value
News	1	DF-4	713	DF-4	791	DF-4	818	DF-4	673
	2	DF-24	377	DF-0	502	DF-11	458	DF-5	491
	3	DF-5	339	EB-87	329	DF-5	406	DF-16	349
	4	EB-72	315	EB-76	303	DF-16	383	DF-36	346
	5	DF-37	288	DF-5	298	EB-109	372	EB-83	327
	6	DF-46	272	DF-32	290	DF-43	371	EB-92	317
	7	DF-47	264	EB-120	282	EB-89	354	DF-12	309
	8	EB-60	246	DF-12	277	EB-92	294	EB-99	300
	9	DF-21	243	EB-98	266	DF-33	265	DF-11	295
	10	EB-148	239	DF-41	263	EB-85	251	DF-45	291

RQ 5.2. In the pie chart, if the proportion of a deviation feature is higher compared to others, it implies that this feature contributes more to the effectiveness of PriCod. Notably, CDF features have the highest total importance value, accounting for 42.4%. This suggests that CDF features, namely Coordinate Deviation Features, contribute the most to the effectiveness of PriCod. The second-highest proportion is attributed to DIF features, accounting for 23.3%.

Answer to RQ5.2: *Both deviation features and embedding features consistently demonstrate their presence among the top-N most influential attributes across a range of subjects. Notably, deviation features exhibit a greater overall importance. Among all the deviation features, the CDF features have the highest contributions to the effectiveness of PriCod compared to other deviation features.*

5.5.6 RQ6: Exploring whether uncertainty-based metrics can enhance the effectiveness of PriCod

Objectives: In the original PriCod, we generate embedding features for each test to indirectly reveal its proximity to the decision boundary. A prior study [9] suggests

that uncertainty-based metrics can also reflect this proximity. Therefore, in this research question, we investigate whether integrating these metrics can enhance PriCod's effectiveness.

Experimental design: To assess whether the integration of uncertainty-based metrics can enhance the effectiveness of PriCod, we incorporate several uncertainty-based metrics into the original PriCod for the purpose of test prioritization. Specifically, we utilize six widely adopted uncertainty-based metrics [9, 6, 88], namely DeepGini, Vanilla SM, PCS, Entropy, Margin, and Least Confidence. The selection of these metrics is based on their widespread use in quantifying uncertainty in the context of DNN testing and their demonstrated effectiveness [101, 9]. The process of constructing the uncertainty feature vector for each test input $t \in T$ is outlined as follows:

- **Calculation of Confidence Scores:** Given a test input t, we compute its uncertainty scores using the aforementioned six uncertainty-based metrics.
- **Generation of Uncertainty Features:** For t, its uncertainty feature vector is generated by concatenating the uncertainty scores obtained from the six metrics. Consequently, for t, PriCod generated a final uncertainty feature vector, denoted as $[S_1, S_2, S_3, S_4, S_5, S_6]$, where each element S_i represents the uncertainty score calculated by the i_{th} uncertainty-based metric.

Finally, for the test t, we integrate its uncertainty features obtained above with embedding features and deviation features (the generation processes for these two types of features can be referred to in Section 5.3.2) to calculate the misclassification probability of this test. We represent this new PriCod method as PriCod$_u$. We compare the effectiveness of PriCod$_u$ and the original PriCod to determine whether integrating the uncertainty-based metrics can enhance PriCod's effectiveness.

Results: The experimental results for RQ6 are presented in Table 5.19. Here, PriCod$_u$ represents the variant of PriCod where uncertainty-based features are incorporated for test prioritization. Notably, for each case, we highlighted the approach with the highest effectiveness in gray. In Table 5.19, we see that the average effectiveness (measured by APFD) of the original PriCod slightly exceeds that of PriCod$_u$. Specifically, the average APFD for PriCod is 0.7810, while for PriCod$_u$, it is 0.7805, with a difference of only 0.0005. PriCod$_u$ demonstrates better effectiveness in a higher proportion of cases, accounting for 70% of all cases. However, in each individual case, the improvement of PriCod$_u$ relative to the original PriCod is slight. For instance, in the case of CIFAR10 with the AlexNet-coreml model, the APFD for PriCod is 0.721, while for PriCod$_u$, it is 0.722. The above experimental results indicate that uncertainty features do not enhance the performance of PriCod in terms of average results. In certain specific subjects, the inclusion of uncertainty features can lead to improvements, but the improvements are minor.

Answer to RQ6: *From the perspective of the average values, uncertainty features do not enhance the performance of PriCod. Although uncertainty features can lead to improvement in some specific subjects, the enhancement is minor.*

Table 5.19: Effectiveness comparison among PriCod and PriCod$_u$ in terms of the APFD values on natural test inputs

Data	Model	Approach	
		PriCod	PriCod$_u$
CIFAR10	AlexNet-coreml	0.721	0.722
	AlexNet-tflite	0.720	0.721
	VGG16-coreml	0.781	0.783
	VGG16-tflite	0.781	0.782
CIFAR100	DenseNet201-coreml	0.788	0.790
	DenseNet201-tflite	0.796	0.789
	ResNet152-coreml	0.765	0.766
	ResNet152-tflite	0.786	0.778
Fashion	LeNet1-coreml	0.815	0.822
	LeNet1-tflite	0.815	0.821
	LeNet5-coreml	0.826	0.832
	LeNet5-tflite	0.824	0.829
Plant	NIN-coreml	0.795	0.798
	NIN-tflite	0.794	0.798
	VGG19-coreml	0.779	0.783
	VGG19-tflite	0.781	0.782
News	GRU-coreml	0.756	0.745
	GRU-tflite	0.757	0.744
	LSTM-coreml	0.771	0.763
	LSTM-tflite	0.770	0.762
Average		0.7810	0.7805

5.6 Discussion

5.6.1 Limitations of PriCod

While PriCod has demonstrated its potential to improve test prioritization for compressed DNN models, it is crucial to recognize its limitations:

[*Model Compression Tools*] PriCod has currently undergone primary testing using two prevalent compression tools, TFLite and CoreML, which are widely utilized in the field. While our method has demonstrated its efficacy with models compressed using these tools, our future endeavors will involve assessing PriCod's performance across a broader spectrum of compression tools.

[*Model Compression Techniques*] Our approach primarily focuses on model compression through quantization, a prevalent method in the field of model compression. However, model compression encompasses a wide range of techniques beyond quantization. In the future, we intend to evaluate PriCod's effectiveness across a broader spectrum of compression methods. This broader assessment will be crucial in ensuring that PriCod remains versatile and capable of addressing various types of compressed DNN models.

[*Model Domains*] PriCod is primarily tailored for test prioritization in image-related domains. While image analysis is a typical application, the applicability of PriCod in other domains remains an area for exploration. In the future, we intend to evaluate PriCod's effectiveness in diverse domains beyond images.

5.6.2 Threats to Validity

Threats to Internal Validity. Internal validity threats primarily arise from the execution of PriCod and its variations, along with the test prioritization approaches being compared. To address this concern, we implement PriCod using the

widely recognized TensorFlow library and the LightGBM framework. Regarding the implementation of the compared approaches, we utilize their original codebases as provided in their respective publications, aiming to minimize potential biases stemming from implementation. Another internal threat emerges from the inherent randomness associated with model training. To mitigate this concern and ensure the reliability of our experimental results, we conducted a statistical analysis by replicating the training procedure ten times. We presented the average experimental outcomes and calculated the statistical significance of the results.

Threats to External Validity. The primary external validity threats originate from the datasets and compressed DNN models employed in our study. To tackle these issues, we encompassed a diverse range of subjects, spanning natural, noisy, and adversarial data. As for the compressed DNN models, we utilized two widely adopted frameworks for model compression: TFLite and CoreML. Our objective is to perform a thorough evaluation of PriCod's effectiveness across various scenarios and to enhance the applicability of our conclusions.

5.7 Related Work

5.7.1 Test prioritization for Deep Neural Networks

In the literature, a variety of techniques have been proposed to prioritize test inputs for Deep Neural Networks (DNNs)[6, 10, 9]. Feng *et al.* [6] introduced DeepGini [6], which identifies potentially misclassified tests by evaluating the model's confidence. This approach is built on the assumption that if a DNN assigns similar probabilities to each class for a test, it is more likely to be incorrectly predicted. Byun *et al.* [110] examined several metrics for prioritizing inputs that reveal software bugs, using white-box measurements of DNN sentiment. These metrics include softmax confidence (i.e., predicted probability for output categories in DNNs using softmax output layers), Bayesian uncertainty (i.e., uncertainty in prediction probability distributions for Bayesian Neural Networks), and input surprise (i.e., the disparity in neuron activation patterns between a test input and training data). Weiss *et al.* [9] investigated the effectiveness of different DNN test input prioritization methods, including notable confidence-based metrics like Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy. Wang et al.[10] introduced PRIMA, a technique for prioritizing test inputs for DNNs through intelligent mutation analysis. PRIMA enhances DNN test prioritization in two significant ways. Firstly, it can be applied not only to classification models but also to regression models. Secondly, PRIMA addresses scenarios where test inputs are generated using adversarial input generation methods[212], which could artificially increase the probability of an incorrect class assignment. However, PRIMA can not be adapted to compressed DNN models since the model mutation rules employed by PRIMA cannot be directly applied to compressed DNN models.

Zheng *et al.* [43] proposed CertPri, a DNN test input prioritization technique that focuses on measuring movement difficulty in the feature space. This method assesses the cost of moving test inputs closer to or farther from the class centers, providing a novel perspective on prioritization strategies. Al-Qadasi *et al.* [213] introduced a new metric, WFDR, for evaluating the effectiveness of prioritizing Dtest in the context of DNNs. The WFDR metric considers fault detection ratio and rate, incorporating adaptive weights to account for prioritization difficulty. This approach offers a

comprehensive assessment framework for prioritization algorithms. Wei *et al.* [44] proposed EffiMAP, an efficient test case prioritization technique utilizing predictive mutation analysis. Without requiring a full mutation analysis, EffiMAP predicts the capability of test cases to expose model prediction failures based on information extracted from the test case execution trace. This pioneering work demonstrates the feasibility of predictive mutation analysis in ranking test cases for deep neural network testing. Tao *et al.* [45] introduced TPFL, a DNN test prioritization technique that employs dynamic spectrum analysis on each neuron. TPFL identifies suspicious neurons causing incorrect decisions in the DNN and prioritizes test inputs based on their ability to activate these neurons. This approach leverages key insights into the relationship between neuron activation and bug-revealing inputs.

5.7.2 Test Prioritization for Traditional Software

In the realm of traditional software testing [214], the concept of test prioritization is a pivotal approach aimed at determining the most efficient sequence for executing test cases, enabling the swift identification of system defects [11, 167, 79, 116, 113, 215]. Rothermel *et al.* [116] pioneered introducing and comparing three distinct test case prioritization methods for regression testing, all rooted in test execution data. Their findings unequivocally showcased that each of the scrutinized prioritization methods substantially heightened the fault detection rate within the test suite. In the pursuit of evaluating the efficacy of diverse test case prioritization techniques in bug detection, Di Nardo *et al.* [79] conducted an empirical case study centered around coverage-based prioritization strategies applied to real-world regression faults. Similarly, Henard *et al.* [77] embarked on an extensive survey, undertaking a meticulous comparison of existing test prioritization approaches. Intriguingly, their investigation revealed marginal disparities between white-box [216, 217, 218, 11] and black-box strategies [219, 220, 221]. Another noteworthy advancement came from Chen *et al.* [167], who introduced the LET method, a pioneering approach to prioritizing test programs in the domain of compiler testing with the primary goal of enhancing efficiency. Chen *et al.* demonstrated the method's efficacy through two interconnected processes. The initial process involves learning, wherein the system identifies distinctive features of test programs and predicts the probability of a new test program uncovering bugs. Subsequently, the scheduling process comes into play, prioritizing test programs based on these predicted probabilities of bug discovery. This innovative dual-process framework proposed by Chen *et al.* constitutes a notable contribution to optimizing compiler testing strategies.

5.7.3 Deep Neural Network Testing

DeepXplore [8] is the first technique targeted at testing DNN models. It proposed neuron coverage, which measures the activation state of neurons, to guide the generation of test inputs. DeepXplore is based on differential testing, and it uses multiple models of a task to detect potential defects. To alleviate the need for multiple models under test, DeepTest [222] leverages metamorphic relations [223] that are expected to hold by a model as its test oracles. Both DeepXplore and DeepTest perturb their test inputs based on the gradient of deep learning models.

The preceding section has discussed test prioritization, which aims to reduce labeling costs and improve DNN testing efficiency. In addition to test prioritization, several test selection approaches have also been proposed to lower labeling costs.

Test selection focuses on accurately estimating the accuracy of the entire dataset by labeling only a selected subset of test inputs. This approach effectively decreases the labeling costs associated with DNN testing. Li *et al.* [48] introduced CES (Cross Entropy-based Sampling), which performs test selection by minimizing the cross-entropy between the selected subset and the complete test set, ensuring that the distribution of the chosen test inputs resembles that of the original test set. Chen *et al.* [46] proposed PACE to guide DNN test selection. Initially, PACE clusters all test inputs into groups based on their testing characteristics. Then, PACE employs the MMD-critic algorithm [49] to select prototypes from each group. For test inputs not falling into any group, PACE utilizes adaptive random testing to select appropriate tests from them.

Wu *et al.* [224] introduced Stratified random Sampling with Optimum Allocation (SSOA), a framework that integrates sampling theory into the task of deep learning test input selection. SSOA leverages stratified random sampling and optimum allocation to provide an unbiased approach for selecting test inputs. This methodology contributes to mitigating biases in the test input selection process, enhancing the representativeness of the chosen inputs. Hao *et al.* [225] proposed Multiple-Objective Optimization-Based Test Input Selection (MOTS) as a method for selecting a more effective test subset to retrain DNN models. In contrast to existing approaches, MOTS considers both the uncertainty of test inputs and the diversity of the test subset. Employing the NSGA-II multiple-objective optimization algorithm, MOTS ensures that the selected test subset exhibits diverse features, providing enhanced support for DNN model retraining.

Wu *et al.* [226] introduced RNNtcs, a method for selecting test cases for Recurrent Neural Networks (RNNs) that combines clustering and uncertainty. RNNtcs aims to identify test cases that can effectively reveal RNN bugs by considering both the clustering structure and uncertainty. This approach is designed to reduce the cost of labeling by focusing on test cases with a higher likelihood of exposing model vulnerabilities. Liu *et al.* [227] proposed DeepState, a test suite selection tool tailored to the specific neural network structures of Recurrent Neural Network (RNN) models. DeepState reduces data labeling and computation costs by selecting data based on a stateful perspective of RNN. This perspective involves identifying potentially misclassified tests by capturing the state changes of neurons in RNN models. DeepState addresses the unique challenges posed by RNN structures in the context of test input selection.

5.7.4 Test Generation approaches for Compressed DNN models

In the literature, several test generation approaches have been proposed for compressed DNN models. Odena and Goodfellow introduced TensorFuzz [228], a pioneering method that utilized coverage-guided fuzzing as a test generation approach. TensorFuzz aimed to reveal difference-inducing inputs between a well-trained DNN and its quantized counterpart. By employing a coverage-guided strategy, TensorFuzz efficiently explored the input space, exposing discrepancies in the behavior of compressed DNN models.

Yahmed *et al.* [229] proposed DiverGet, a search-based testing framework designed specifically for quantization assessment in compressed DNN models. DiverGet introduces a structured space of metamorphic relations that simulate natural distortions on input data. These metamorphic relations are then systematically explored

to optimize the revelation of disagreements among DNNs subjected to different arithmetic precision. By defining and strategically navigating this metamorphic space, DiverGet provides a comprehensive approach to evaluating the impact of quantization on DNN models.

Xie *et al.* [230] proposed DiffChaser, a novel automated black-box disagreement detection technique tailored for multiple variants of a DNN. The core premise behind DiffChaser is the identification of similarities in decision boundaries between a DNN and its quantization-aware training (QC) version variants. The rationale is that the decision boundaries tend to exhibit resemblance, particularly in proximity to the boundary itself. Consequently, inputs near these decision boundaries are more likely to capture the discrepancies in decision boundaries, representing the disagreement among the DNN models. DiffChaser leverages prediction uncertainty as a guiding metric and automatically generates inputs that lie in the vicinity of decision boundaries to unveil the distinctions between DNN variants.

5.8 Conclusion

To address the challenge of labeling-cost reduction in the context of testing compressed DNN models, we proposed PriCod, a novel test prioritization approach designed to identify and prioritize potentially misclassified tests. PriCod is rooted in two fundamental premises: firstly, that significant prediction deviations between compressed and original DNN models signify a greater likelihood of test input misclassification, and secondly, that test inputs situated near decision boundaries are more susceptible to misclassification. Building upon these premises, PriCod generates two distinct feature types for each test input for the purpose of test prioritization: deviation features, quantifying the prediction deviation caused by model compression, and embedding features, which indirectly reflect proximity to decision boundaries by leveraging intrinsic information about test inputs. These features are combined to calculate the misclassification probability of each test input. Subsequently, PriCod ranks all tests in descending order based on their misclassification probability. We conducted a comprehensive assessment of PriCod's performance, using different types of test inputs and various test prioritization techniques. Our findings consistently showcased PriCod's superior performance, revealing an average improvement from 7.43% to 55.89% for natural test inputs, 7.92% to 52.91% for noisy inputs, and 7.03% to 51.59% for adversarial inputs compared to existing methods.

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

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

6 Test Input Prioritization for 3D Point Clouds

In this chapter, we propose PCPrior, a novel test prioritization method tailored for 3D point clouds. The core idea behind PCPrior is that test input close to the decision boundary of the model is more likely to be misclassified. By identifying and prioritizing such potentially misclassified test inputs, developers can allocate limited label budgets more effectively and accelerate the debugging process.

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

- Yinghua Li, Xueqi Dang, Lei Ma, Jacques Klein, Yves LE Traon and Tegawendé F. Bissyandé. Test Input Prioritization for 3D Point Clouds. ACM Transactions on Software Engineering and Methodology(TOSEM). Accepted for publication on Jan. 15, 2024.

Contents

6.1 Introduction

The advent of point cloud data has revolutionized various fields, such as computer vision [1, 2], autonomous driving [3, 4], augmented reality [231, 232, 233] and smart cities [234, 234], by enabling highly accurate and detailed representation of real-world environments. A Point cloud [17] refers to a collection of three-dimensional data points in space, typically representing the surface geometry or shape of real-world objects or environments. Each data point in a point cloud is defined by its spatial coordinates (x, y, z) and, in some cases, additional attributes such as color or intensity values. Figure 6.1 illustrates an example of a point cloud representing the shape of a car, composed of thousands of individual points. To showcase the inherent three-dimensional attributes of the point cloud, we present multiple perspectives of the object from different viewing angles. From specific angles, the car is easily identifiable and recognizable. However, from some angles, it becomes challenging to identify the object as a car. Point clouds are commonly generated using various sensing technologies, including LiDAR (Light Detection and Ranging) [35], depth cameras [235], or structured light scanners [236], which capture the physical measurements of points in the environment.

Compared to two-dimensional data like images, 3D point clouds have inherent differences and significant advantages. First, 3D point clouds offer a three-dimensional depiction of objects, resulting in higher accuracy and reliability when identifying complex 3D shapes and volumes. Moreover, point cloud data can directly capture surface details and morphology of objects, making them difficult to be replaced by images in many practical applications. Consequently, the integration of point cloud processing in safety-critical applications, such as autonomous driving [18, 4], medical imaging [237], and industrial automation [238], has become increasingly prevalent. For instance, 3D point clouds can be utilized for autonomous driving in the context of obstacle detection and perception [239]. More specifically, 3D point cloud data obtained from LiDAR (Light Detection and Ranging) sensors [35] can provide a rich and detailed representation of the surrounding environment in three-dimensional space. By leveraging this data, it becomes feasible to identify and localize various objects on the road, such as automobiles, pedestrians, cyclists, and obstacles. Leveraging these 3D data, autonomous driving systems can employ 3D classification models to detect and categorize objects, thereby guiding the avoidance of obstacles. Hence, the accuracy of these 3D classification models plays a pivotal role in ensuring the safety of autonomous driving.

In recent years, Deep Neural Networks (DNNs) have emerged as a powerful tool for various computer vision tasks [240, 241], and their application to 3D point cloud data has garnered significant attention. Ensuring the reliability of DNNs operating on point cloud data is crucial for safe and efficient functioning. DNN testing [9, 111] has become a widely adopted approach to assess and ensure the quality of such networks. Nevertheless, prior investigations [46, 6, 10] have highlighted a central challenge pertaining to DNN testing: the significant cost incurred in labeling test inputs to verify the accuracy of DNN predictions. First, the scale of the test set is typically extensive. Second, manual labelling is mainstream, typically necessitating the involvement of multiple annotators to ensure the accuracy and consistency of the labeling process for each test input.

The challenges are further compounded in the case of 3D point cloud data.

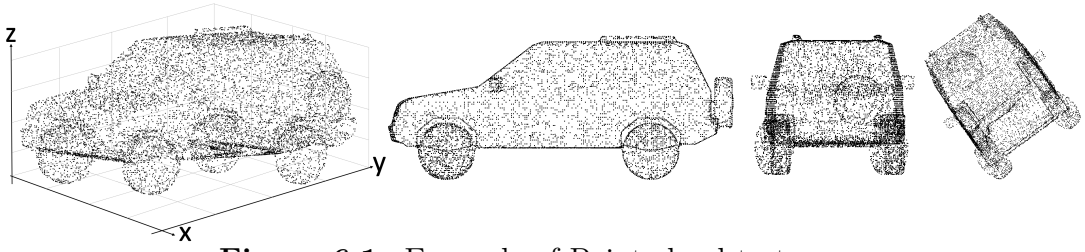


Figure 6.1: Example of Point cloud test cases

In addition to the aforementioned obstacles, labeling point cloud data presents additional distinctive challenges compared to traditional image/text data.

- **Data representation** Image data is represented as two-dimensional matrices, with each pixel having a distinct position and value. In contrast, point cloud data comprises an unordered set of points, each possessing three-dimensional coordinates and additional attributes such as color and normals. This distinctive data representation significantly increases the complexity of labeling, necessitating additional processing and interpretation steps.
- **Sparsity of point clouds** Point cloud data is generally characterized by sparsity compared to image data. There can be missing points or noise in the point cloud, and the distribution of points is non-uniform. This inherent sparsity poses challenges for accurate labeling.
- **Expert knowledge for 3D point clouds** Labeling 3D point cloud data necessitates domain-specific expertise due to its unique characteristics. With a large number of three-dimensional points, each with its own coordinates and potential attributes, accurately labeling 3D point cloud data requires expert knowledge. This expertise is crucial for understanding and interpreting the geometric attributes, shapes, and potentially semantic information conveyed by the points.

To address the issue of labeling cost in the context of DNNs, previous research efforts [6] have primarily focused on test prioritization, which aims to prioritize test inputs that are more likely to be misclassified by the model. By allocating resources to label these challenging inputs first, developers can ensure priority for critical test cases, ultimately resulting in reduced overall labeling costs. Existing test prioritization approaches [10, 6, 9] can be broadly categorized into two main groups: coverage-based and confidence-based. Coverage-based techniques prioritize test inputs based on the coverage of neurons [11, 12]. In contrast, confidence-based approaches operate under the assumption that test inputs for which the model exhibits lower confidence are more likely to be misclassified. Notably, confidence-based approaches have been demonstrated to be more effective than coverage-based approaches in the existing studies [6]. Weiss *et al.* [9] conducted a comprehensive exploration of diverse test input prioritization techniques, encompassing several confidence-based metrics that can be adapted to 3D point cloud data, such as DeepGini, Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy.

Although the confidence-based test prioritization approaches have demonstrated efficacy in specific contexts such as image and text data, they encounter several limitations when applied to 3D point cloud data.

- **Noises in 3D point cloud data** 3D point cloud data can exhibit inherent noise, which arises from various sources such as sensor noise and non-uniform sampling density. These noise factors can affect the effectiveness of confidence-based approaches. Specifically, in the presence of noise, the model can erroneously assign

a high probability to an incorrect category for a given test sample. Consequently, confidence-based approaches incorrectly assume that the model is highly confident of this particular test, considering it will not be misclassified. However, the model's prediction on this test sample is indeed incorrect (i.e., this test is misclassified by the model).

- **Missing crucial spatial features** Confidence-based methods typically rely on the model's prediction confidence on test samples. However, in the case of 3D point cloud data, the point cloud exhibits complex spatial characteristics, and relying solely on the confidence feature of the model's prediction for test prioritization is limited. In other words, confidence-based methods fail to fully leverage the informative features inherent in point cloud data for test prioritization.

In addition to coverage-based and confidence-based techniques, Wang *et al.* [10] proposed PRIMA, which leverages mutation analysis and learning-to-rank methodologies for test input prioritization in DNNs. However, although demonstrating effectiveness in the domain of DNN test prioritization, PRIMA faces challenges when applied to 3D point cloud data. The reason is that: 1) the mutation operators utilized in PRIMA are primarily designed for two-dimensional images, text, and predefined features. These operators are not directly applicable to 3D point cloud data. In contrast to conventional image or text data, 3D point clouds exhibit a distinctive three-dimensional representation characterized by a substantial quantity of points; 2) even when considering the possibility of utilizing dimensionality reduction techniques to transform 3D data into two-dimensional images and integrating them into PRIMA, practical issues emerge. The execution flow of PRIMA necessitates the mutated two-dimensional images to be fed into the evaluated model for comparing the prediction results between mutants and original inputs. However, the model employed for 3D point clouds is inherently tailored to process three-dimensional data and lacks the capability to classify the mutated two-dimensional images. As a result, even in scenarios where dimensionality reduction tools are accessible, PRIMA remains unsuitable for accommodating 3D point cloud data.

In this paper, we propose PCPrior (3D **P**oint **C**loud Test **Prior**itization), a novel test prioritization approach specifically designed for 3D point cloud test cases. PCPrior leverages the unique characteristics of 3D point clouds to prioritize tests. It is crucial to emphasize that our approach focuses on datasets where each 3D point cloud corresponds to an individual test case. Therefore, each test case is constituted by a collection of points. The core idea behind the PCPrior framework is that: test inputs situated closer to the decision boundary of the model are more likely to be predicted incorrectly, which has been proven in the prior research [20]. PCPrior aims to prioritize such possibly-misclassified tests higher.

To reflect the distance between a test (a point cloud) and the decision boundary, we adopt a vectorization approach to map each test to a low-dimensional space, indirectly revealing the proximity between the point cloud data and the decision boundary. Based on this vectorization strategy, we design a diverse set of features to characterize a point cloud test, including Spatial Features (SF), Mutation Features (MF), Prediction Features (PF), and Uncertainty Features (UF). Notably, SF and MF are specifically designed based on the characteristics of point clouds. Specifically, these features play a pivotal role in capturing essential aspects, including the spatial properties of the point cloud, mutation information present in the input, predictions generated by the DNN model, and the corresponding confidence levels. PCPrior

constructs a comprehensive feature vector through the concatenation of these four feature types and leverages a ranking model to learn from it for effective test prioritization.

Compared to existing test prioritization approaches, PCPrior has the following advantages:

- **Tailored for 3D Point Cloud Data** PCPrior is specifically designed to address the challenges of test prioritization for 3D point cloud data. Unlike existing approaches that focus on 2D images or text data, PCPrior leverages the distinctive characteristics of 3D point clouds and provides a more targeted approach for prioritizing tests.
- **Effective Utilization of Spatial Features** PCPrior leverages the spatial features of 3D point clouds, which are essential for understanding the geometric attributes and shapes of objects in the data. Unlike confidence-based approaches that solely rely on prediction confidence, PCPrior incorporates spatial features into the prioritization process. By considering the spatial properties of the point cloud data, PCPrior can effectively capture the informative features necessary for accurate test prioritization.
- **Comprehensive Feature Generation Mechanism** In addition to incorporating spatial characteristics, PCPrior integrates confidence-based features while also taking into account mutation and prediction features. By combining these features into a comprehensive feature vector, PCPrior captures a rich set of information that enhances the effectiveness of test prioritization.

PCPrior exhibits broad applicability across diverse domains. As a case in point, in the field of autonomous driving, when testing a 3D shape classification model, the utilization of sensors facilitates the collection of unlabeled test sets comprising surrounding 3D point clouds. PCPrior can be utilized to identify and prioritize test instances that are more likely to be misclassified by the model. By focusing on labeling these possibly-misclassified test inputs, it results in a reduction of both labeling time and the manual efforts involved in the labeling process.

To evaluate the effectiveness of PCPrior, we conduct an extensive experimental evaluation on a diverse set of 165 subjects, encompassing both natural datasets and noisy datasets. We compare PCPrior with several existing test prioritization approaches that have demonstrated effectiveness in prior studies [6, 9]. The evaluation metrics include the Average Percentage of Fault-Detection (APFD) [11] and Percentage of Fault Detected (PFD) [6], which are standard and widely-adopted metrics for test prioritization. The experimental results demonstrate the superiority of PCPrior over existing test prioritization techniques. Specifically, when applied to natural datasets, PCPrior consistently outperforms all the comparative test prioritization approaches, yielding an improvement ranging from 10.99% to 66.94% in terms of APFD. Moreover, on noisy datasets, the improvement ranges from 16.62% to 53%. We publish our dataset, results, and tools to the community on Github[1].

Our work has the following major contributions:

- **Approach** We propose PCPrior, the first test prioritization approach specifically for 3D point cloud data. To this end, we design four types of features that can comprehensively extract information from a 3D point cloud test. We employ effective ranking models to learn from the generated features for test prioritization.
- **Study** We conduct an extensive study based on 165 3D point cloud subjects

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

involving natural and noisy datasets. We compare PCPrior with multiple test prioritization approaches. Our experimental results demonstrate the effectiveness of PCPrior.
- **Performance Analysis** We compare the contributions of different types of features to the effectiveness of PCPrior. We also investigate the impact of main parameters in PCPrior.

6.2 Background

6.2.1 Deep Learning for 3D Point Clouds

The rapid advancements in sensor technologies, such as LiDAR (Light Detection and Ranging) [35] and RGB-D (Red-Green-Blue Depth) cameras [36], have led to the proliferation of three-dimensional (3D) point cloud data. These representations find significant utility in various fields, including medical treatment [242], autonomous driving [243, 18], and robotics [244, 245]. Typically, a point cloud represents a collection of data points in 3D space, each point typically denoted by its spatial coordinates (x, y, z) and, in some cases, additional attributes like color or intensity values [37]. Figure 6.1 illustrates one concrete example of a point cloud, showcasing the shape of a car. Each point cloud comprises numerous individual points.

The emergence of Deep Learning [246, 70], particularly Convolutional Neural Networks (CNNs) and PointNet [38], has revolutionized the analysis and understanding of 3D point cloud data. Moreover, the availability of numerous publicly accessible datasets, such as ModelNet [39], ShapeNet [40], and S3DIS [41], has played a pivotal role in stimulating research endeavors focused on deep learning techniques applied to 3D point clouds. This surge in research has led to the development of numerous methods addressing various problems in point cloud processing. One extensively studied problem in this domain is **three-dimension (3D) shape classification**, which focuses on utilizing DNNs to classify three-dimensional shapes. For example, in the field of autonomous driving, 3D shape classification can be utilized to categorize various objects on the road, such as vehicles, pedestrians, traffic signs, etc. By accurately classifying these objects, the autonomous driving system can better understand the surrounding environment, enabling more precise decision-making.

3D shape classification typically involves three main steps: **1) Learning individual point embeddings** Initially, each point in the point cloud undergoes processing to acquire its embedding representation. **2) Obtaining global shape embedding** Subsequently, these individual point embeddings are aggregated to generate the global shape embedding for the entire point cloud. This step aims to capture the overall structure and shape characteristics of the entire point cloud. **3) Classification Processing** Finally, the global shape embedding is input into several fully connected layers for classification. These layers are responsible for determining the category of the 3D shape represented by the point cloud based on the extracted global features.

In the literature [17, 16, 38, 42], several approaches have been proposed to tackle the challenge of 3D shape classification, such as PointConv [17], Dynamic Graph Convolutional Neural Network (DGCNN) [16], and PointNet [38]. PointConv is a specialized convolutional neural network designed for processing 3D point clouds. Training multi-layer perceptrons on local point coordinates enables the construction of deep networks directly on 3D point clouds for efficient analysis. DGCNN, tailored

for 3D point cloud data, leverages intrinsic spatial relationships by modeling them as graphs. Through graph convolutions and dynamic adaptation of the graph structure based on input data, DGCNN effectively learns and processes point cloud representations. PointNet, a widely adopted architecture for 3D point cloud data, incorporates a shared multi-layer perceptron (MLP) with max-pooling for local feature extraction and a symmetric function for aggregating global features. T-Net layers enable PointNet to learn transformation matrices, enhancing its robustness to input variations. PointNet has demonstrated impressive capabilities in 3D shape classification.

6.2.2 Mutation Testing

Mutation testing in traditional software engineering In the field of software testing [185, 78, 184], mutation testing [152, 111] presents a robust methodology for evaluating the effectiveness of a test suite in identifying code defects. The primary objective of mutation testing revolves around assessing the test suite's capacity to detect and localize faults within the code. The fundamental premise is that: if a test case can successfully uncover a mutation, thereby revealing a discrepancy in program behavior compared to the original code, it signifies the test case's potential to identify bugs in real-world scenarios. These mutations are intentionally introduced into the original program through simple syntactic modifications, resulting in the creation of a set of defective programs known as mutants, each possessing a distinct syntactic alteration. To assess the efficacy of a given test suite, these mutants are executed using the input test set, allowing an examination of whether the injected faults can be detected.

The process of mutation testing, as delineated in prior research [129], entails generating a set of mutated programs, denoted as p', by applying predefined mutation rules to an original program P. These mutations introduce minor modifications to P, thereby creating a collection of mutants for evaluation. The determination of whether a mutant p' is classified as "killed" or "survived" is contingent upon the disparity observed in the test result between p' and the original program. More specifically, a mutant is categorized as "killed" if the test case yields a different behavior compared to that of the original test. The killing of a mutant indicates that the corresponding test case has successfully identified and flagged a potential defect in the code under examination. This discrepancy in behavior suggests that the test case has effectively detected and indicated the presence of a possible defect within the code. Conversely, a mutant is regarded as "survived" if the test result remains unchanged in comparison to the original program, suggesting that the test case fails to uncover the introduced fault. When a test suite is able to "kill" many mutants, it indicates that the suite has a higher capability to detect and localize faults within the code.

Mutation testing for DNNs Researchers have introduced various approaches and tools aimed at adapting mutation testing for deep learning systems [20, 59, 61]. Notable contributions include DeepMutation [20], DeepMutation++ [59], MuNN [61], and DeepCrime [60]. DeepMutation [20] is designed to evaluate the quality of test data for deep learning (DL) systems using mutation testing. This innovative approach encompasses the creation of mutation operators at both the source and model levels, strategically introducing faults into various components such as training data, programs, and DL models. The evaluation of test data effectiveness is subsequently conducted by analyzing the detection of these introduced faults. Building upon

this foundation, DeepMutation++[59] represents an advanced iteration, introducing innovative mutation operators tailored for feed-forward neural networks (FNNs) and Recurrent Neural Networks (RNNs). Notably, it possesses the capability to dynamically mutate the run-time states of an RNN. Shen *et al.* proposed MuNN [61], an intricate mutation analysis method designed explicitly for neural networks. The method establishes five mutation operators, each rooted in the distinctive characteristics of neural networks. In a remarkable stride towards practical application, Humbatova *et al.* introduced DeepCrime [60], a mutation testing tool that implements DL mutation operators based on real-world DL faults. Furthermore, Jahangirova *et al.* [111] conducted a comprehensive empirical study of DL mutation operators found in the existing literature. Their study, which includes 20 DL mutation operators such as activation function removal and layer addition, suggests that while most operators are useful, their configuration needs careful consideration to avoid rendering them ineffective.

6.2.3 Test Input Prioritization for DNNs

Test prioritization [113] is a critical process in software testing that seeks to establish an optimal sequence for unlabelled tests. Its core objective is to identify and prioritize potentially misclassified tests, enabling their early labelling and consequently leading to a reduction in the overall labelling cost. The majority of approaches for prioritizing tests in Deep Neural Networks (DNNs) [10, 6, 108] can be categorized into two main groups: coverage-based and confidence-based [10]. Coverage-based approaches, exemplified by CTM [11], involve the direct extension of conventional software system testing methods to the domain of DNN testing. In contrast, confidence-based approaches prioritize test inputs based on the model's level of confidence. Specifically, these methods aim to identify inputs that are likely to be misclassified by the DNN model, as indicated by the model assigning similar probabilities to each class. DeepGini [6] stands as a classic confidence-based test prioritization method that has been empirically shown to outperform existing coverage-based techniques in terms of both effectiveness and efficiency. Other confidence-based test prioritization methods, such as Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy, have also been evaluated in recent research [9]. These metrics have demonstrated efficacy in identifying potentially misclassified test inputs and can assist in guiding test prioritization efforts.

While confidence-based methods can be applied to 3D point cloud data, they have certain limitations. 3D point cloud data is typically characterized by its large-scale and highly detailed nature, typically consisting of millions or even billions of points. However, confidence-based methods, when prioritizing tests, primarily focus on the uncertainty associated with the model's classification of the test inputs, neglecting the intrinsic raw feature information contained within the point cloud data. Furthermore, point cloud data is prone to noise, which can adversely impact the reliability of confidence scores assigned by these approaches. In the presence of noise, confidence-based methods can exhibit high confidence in incorrect labels. Consequently, in such cases, tests that will be misclassified by the model are mistakenly assigned inappropriate confidence scores, resulting in them not being prioritized higher. These factors collectively contribute to the diminished performance of confidence-based methods in the context of point cloud data.

In addition to the aforementioned test prioritization methods, PRIMA [10],

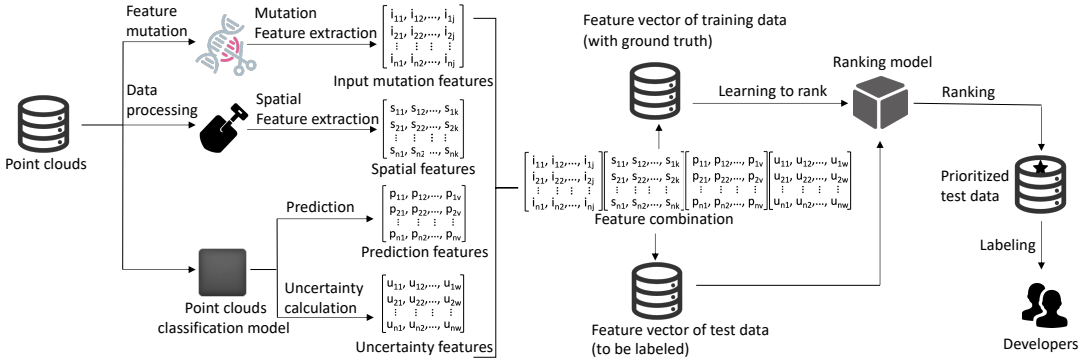


Figure 6.2: Overview of PCPrior

proposed by Wang *et al.*, employs mutation analysis to prioritize test inputs that can uncover faults. However, point cloud data represents unstructured sets of points in three-dimensional space, which makes the mutation rules of PRIMA not adapted.

6.3 Approach

6.3.1 Overview

In this paper, we introduce PCPrior, a novel approach tailored for test prioritization in the domain of 3D point cloud data. The overview of PCPrior is depicted in Figure 6.2, which provides a visual representation of its key components. Specifically, when given a test set T targeted at a DNN model M, we outline the fundamental workflow of PCPrior as follows. A more comprehensive exposition of this workflow is presented in subsequent sections.

- **Feature Generation:** In the initial stage, PCPrior generates four distinct types of features for each test $t \in T$, which are purposefully designed to capture the characteristics of 3D point cloud data. These four types of features encompass Spatial Features, Mutation Features, Prediction Features, and Uncertainty Features. In Figure 6.2, the matrices represent features generated from the test inputs. Here, n denotes the number of test inputs in the test set T. Since there are n tests in T, each matrix has n rows. Each row in the matrix represents a feature vector generated for a specific test. From top to bottom, the first matrix illustrates the input mutation features for all tests in the test set, with dimensions $n \times j$, where j denotes the number of mutation features for each test. The second matrix represents spatial features for all tests in the test set, having dimensions $n \times k$, where k signifies the number of spatial features for each test. The third matrix showcases prediction features, having dimensions $n \times v$, where v represents the number of prediction features for each test. The fourth matrix displays uncertainty features, with dimensions $n \times w$, where w denotes the number of uncertainty features for each test. For example, in the matrix of spatial features, $\{s_{21}, s_{22}, \ldots, s_{2k}\}$ represent all spatial features generated for the second test input in T. In Sections 6.3.2 to 6.3.5, we provide a detailed explanation of the meaning, generation methods, and motivations behind each feature type.
- **Feature Concatenation:** For each test $t \in T$, PCPrior has generated four types of features in the previous step. In this step, PCPrior concatenates these four types of features, resulting in the generation of the final feature vector specifically associated with the test t. In particular, the process is depicted in Figure 6.2 where four matrices are concatenated, forming a large matrix with dimensions of $n \times (j + k + v + w)$.

- **Learning to Rank:** PCPrior takes the final feature vector of each test $t \in T$ and inputs it into a pre-trained ranking model, specifically LightGBM [89]. The ranking model automatically learns the probability of misclassification for each test based on its feature vector. PCPrior leverages these probabilities to sort the tests, placing those with a higher probability of being misclassified by the model at the forefront.

6.3.2 Spatial Feature Generation

Based on the test set T, we generated six types of spatial features from each point cloud test input, including variance [247], mean [248], median [248], scale [249], skewness [250], and kurtosis [250]. We provide detailed explanations of each feature below. PCPrior leverages the spatial features of tests to identify their spatial proximity. As illustrated in Figure 6.2, prior to the generation of spatial features, a data processing step is executed. This step encompasses the reading and transformation of the point cloud dataset. Upon accessing the point cloud data, the coordinates of each point within the point cloud (commonly represented as x, y, z coordinates), along with any supplementary attributes (such as color and intensity), are transformed into a numpy array format.

The rationale behind generating these features stems from the observation made by Ma *et al.* [20] that misclassified inputs typically locate near the decision boundary of a DNN model. In light of this observation, our approach entails the generation of a diverse set of spatial features from each test input, effectively capturing its unique characteristics. As a result, each test instance is transformed into a spatial feature vector, indirectly reflecting the test's proximity to the decision boundary. Tests that exhibit closer proximity to the decision boundary are considered more susceptible to being predicted incorrectly. Motivated by this insight, PCPrior utilizes the spatial features of test inputs to assess their probability of being misclassified. For a given point cloud P, Formula 6.1 illustrates the process of generating its spatial features (SF).

$$V_{SF} = Concat(\sigma^2(P), \mu(P), Median(P), Scale(P), Skewness(P), Kurtosis(P)) \tag{6.1}$$

In Formula 6.1, all feature computations rely on the coordinates of points in the point cloud P along the three coordinate axes (x, y, z). $V_{spatial}$ represents the resulting spatial feature vector for the point cloud P. Below, we use variance features as a specific example to clarify the calculation process for each type of spatial feature. Assuming P consists of five points with coordinates along the x, y, and z axes denoted as $[x_1, x_2, x_3, x_4, x_5]$, $[y_1, y_2, y_3, y_4, y_5]$, and $[z_1, z_2, z_3, z_4, z_5]$ respectively, the variance for the x-axis is calculated as $\text{var}([x_1, x_2, x_3, x_4, x_5]) = 0.5$, for the y-axis as $\text{var}([y_1, y_2, y_3, y_4, y_5]) = 0.3$, and for the z-axis as $\text{var}([z_1, z_2, z_3, z_4, z_5]) = 0.8$. Consequently, the final variance feature vector of P is $[0.5, 0.3, 0.8]$. Similar computations are performed for other types of spatial features. Specifically, in Formula 6.1, $\sigma^2(P)$ represents the variance features of all points in the point cloud P along each coordinate axis. $\mu(P)$ represents the mean features, Median(P) denotes the median features, Scale(P) represents scale features, Skewness(P) indicates the skewness features, and Kurtosis(P) corresponds to the kurtosis features.

- **Variance Features** [247] Variance features serve as statistical indicators for measuring the variability or dispersion of points within a dataset. They quantify the variances of point cloud data along each coordinate axis, thereby providing

crucial information about the spatial distribution of points. Given a point cloud P consisting of N points, where the x, y, z coordinates of each point are respectively $X = [x_1, x_2, \ldots, x_n]$, $Y = [y_1, y_2, \ldots, y_n]$, and $Z = [z_1, z_2, \ldots, z_n]$, Formula 6.2 illustrates the computation process for the variance feature vector of the point cloud P.

$$\sigma^2(P) = [var(X), var(Y), var(Z)] \tag{6.2}$$

where $var(X)$ represents the variance of the X-coordinates of all points in the point cloud P. Namely, it is the variance of $X = [x_1, x_2, \ldots, x_n]$. $var(Y)$ represents the variance of $Y = [y_1, y_2, \ldots, y_n]$, and $var(Z)$ represents the variance of $Z = [z_1, z_2, \ldots, z_n]$. Formula 6.3 precisely illustrates the computation process for the variance of X-coordinates. The procedures for computing the variance of Y and Z coordinates follow a similar approach.

$$var(X) = \frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2 \tag{6.3}$$

where $var(X)$ represents the variance of the X-coordinates of all points in P. Specifically, it is the variance of $X = [x_1, x_2, \ldots, x_n]$. μ represents the mean of the X-coordinates of all points in P. N denotes the total number of points in P. x_i represents the X-coordinates of the i-th point in P.
Specifically, the utilization of variance features in point cloud analysis offers notable benefits: **1) Quantifying dispersion** Variance features enable a quantitative assessment of the dispersion of point cloud data along different coordinate axes. Larger variance values indicate a more scattered distribution of points along the corresponding axis, while smaller variance values suggest a more concentrated distribution. These insights are essential for comprehending the spatial characteristics and shape of the point cloud. **2) Extracting shape information** Variance features facilitate the extraction of rough shape information from the point cloud. By comparing the variances along different coordinate axes, conclusions can be drawn regarding the extension or distribution of the point cloud in various directions. For instance, if the variance along a particular axis significantly surpasses that of the other axes, it implies a greater extension of the point cloud's shape in that specific direction.

- **Mean Features** [248] In the context of 3D point cloud data, mean features refer to the feature values obtained by averaging the attributes (such as coordinates, normals, etc.) of each point in the point cloud. They represent the average attributes of the entire point cloud and provide information about the overall shape or other properties.
  Given a point cloud P consisting of N points, with the x, y, and z coordinates of each point represented as $X = [x_1, x_2, \ldots, x_n]$, $Y = [y_1, y_2, \ldots, y_n]$, and $Z = [z_1, z_2, \ldots, z_n]$, Formula 6.4 demonstrates the calculation process for the mean feature vector of the point cloud P.

$$\mu(P) = [mean(X), mean(Y), mean(Z)] \tag{6.4}$$

where $mean(X)$ denotes the mean of $X = [x_1, x_2, \ldots, x_n]$, $mean(Y)$ represents the mean of $Y = [y_1, y_2, \ldots, y_n]$, and $mean(Z)$ signifies the mean of $Z = [z_1, z_2, \ldots, z_n]$. Formula 6.5 details the computation process for the mean of X-coordinates. The

procedures for calculating the mean of Y and Z coordinates follow a similar approach.

$$mean(X) = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{6.5}$$

where $mean(X)$ represents the mean of the X-coordinates of all points in P. Specifically, it is the mean of $X = [x_1, x_2, \ldots, x_n]$. N denotes the total number of points in P. x_i represents the X-coordinates of the i-th point in P.
We utilize mean features in test prioritization for the following reasons: **1) Comprehensive nature** Mean features consolidate the information of the entire point cloud into a single feature vector, offering comprehensive insights about the overall characteristics. Such comprehensive features facilitate a rapid understanding of the global properties of the point cloud. **2) Dimensionality reduction** Point cloud data typically comprise a large number of points, each potentially possessing multiple attributes. By employing mean features, the point cloud data can be reduced from a high-dimensional space to a lower-dimensional feature vector, thereby reducing computational complexity and memory consumption.

- **Median Features** [248] In the context of 3D point cloud data, median features pertain to the feature values obtained by calculating the median of the coordinate attributes (X, Y, Z) within the point cloud. They serve as indicators of the central tendency of attribute values within the point cloud.
Given a point cloud P comprising N points, where the x, y, and z coordinates of each point are denoted as $X = [x_1, x_2, \ldots, x_n]$, $Y = [y_1, y_2, \ldots, y_n]$, and $Z = [z_1, z_2, \ldots, z_n]$, Formula 6.6 elucidates the computation process for the median feature vector of the point cloud P.

$$Median(P) = [median(X), median(Y), median(Z)] \tag{6.6}$$

where $median(X)$ refers to the median of the X-coordinates of all points in the point cloud P (i.e., $X = [x_1, x_2, \ldots, x_n]$), $median(Y)$ represents the median of $Y = [y_1, y_2, \ldots, y_n]$, and $median(Z)$ signifies the median of $Z = [z_1, z_2, \ldots, z_n]$. Formula 6.7 precisely outlines the computation process for the median of X-coordinates. The procedures for calculating the median of Y and Z coordinates follow a similar approach.

$$median(X) = \begin{cases} X\left(\frac{N+1}{2}\right) & \text{if } N \text{ is odd} \\ \frac{X\left(\frac{N}{2}\right)+X\left(\frac{N}{2}+1\right)}{2} & \text{if } N \text{ is even} \end{cases} \tag{6.7}$$

where median(X) represents the median of the X-coordinates of all points in P. Specifically, it is the median of $X = [x_1, x_2, \ldots, x_n]$. N denotes the total number of points in P. $X\left(\frac{N+1}{2}\right)$ denotes the value in X located at the middle position. $X\left(\frac{N}{2}\right)$ and $X\left(\frac{N}{2}+1\right)$ represent the two values in X located at the middle positions when N is even.
The utilization of median features is motivated by the following factors: 1) Median features exhibit reduced sensitivity to outliers compared to mean features, rendering them more reliable and capable of providing more accurate representations in the presence of extreme values. 2) Median features demonstrate heightened stability by being less influenced by variations in attribute value distributions, thereby facilitating a more consistent representation of the point cloud data.

- **Scale Features** [249] In the context of 3D point cloud data, scale features refer to the differences between the minimum and maximum values of each point in the three dimensions (X, Y, and Z) of the point cloud. Range features can provide information about the scale of the point cloud data, specifically the spatial extent of the point cloud in each dimension.
  Given a point cloud P consisting of N points, where the x, y, and z coordinates of each point are represented as $X = [x_1, x_2, \ldots, x_n]$, $Y = [y_1, y_2, \ldots, y_n]$, and $Z = [z_1, z_2, \ldots, z_n]$, Formula 6.8 illustrates the computation process for the scale feature vector of the point cloud P.

$$Scale(P) = [scale(X), scale(Y), scale(Z)] \tag{6.8}$$

  where $scale(X)$ denotes the scale of the X-coordinates of all points in the point cloud P (i.e., $X = [x_1, x_2, \ldots, x_n]$), $scale(Y)$ represents the scale of the Y-coordinates (i.e., $Y = [y_1, y_2, \ldots, y_n]$), and $scale(Z)$ signifies the scale of the Z-coordinates (i.e., $Z = [z_1, z_2, \ldots, z_n]$). Formula 6.9 precisely delineates the computation process for the scale of X-coordinates. The procedures for calculating the scale of Y and Z coordinates follow a similar approach.

$$scale(X) = max(X) - min(X) \tag{6.9}$$

  where $scale(X)$ represents the scale of the X-coordinates of all points in P. $max(X)$ represents the maximum value in $X = [x_1, x_2, \ldots, x_n]$. $min(X)$ represents the minimum value in it.
  The utilization of scale features lies in that they serve as descriptive features of the point cloud data, providing an overall characterization of the spatial attributes of the point cloud.
- **Skewness Features** [250] Within the context of 3D point cloud data, the skewness feature is a statistical measure employed to quantify the degree of skewness in the distribution of data. It assesses the extent to which the point cloud data distribution deviates from symmetry. In a point cloud P consisting of N points, with the x, y, and z coordinates of each point denoted as $X = [x_1, x_2, \ldots, x_n]$, $Y = [y_1, y_2, \ldots, y_n]$, and $Z = [z_1, z_2, \ldots, z_n]$, Formula 6.10 illustrates the computation process for the skewness feature vector of the point cloud P.

$$Skewness(P) = [skewness(X), skewness(Y), skewness(Z)] \tag{6.10}$$

  where $skewness(X)$ denotes the skewness of $X = [x_1, x_2, \ldots, x_n]$, $skewness(Y)$ represents the skewness of $Y = [y_1, y_2, \ldots, y_n]$, and $skewness(Z)$ signifies the skewness of $Z = [z_1, z_2, \ldots, z_n]$. Formula 6.11 outlines the computation process for the skewness of the X-coordinates. The procedures for calculating the skewness of the Y and Z coordinates follow a similar approach.

$$skewness(X) = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{x_i - \mu}{\sigma} \right)^3 \tag{6.11}$$

  where μ represents the mean of $X = [x_1, x_2, \ldots, x_n]$, and σ denotes the standard deviation of X. N denotes the total number of points in P, and x_i represents the X-coordinates of the i-th point in P.
  The utilization of the skewness feature is motivated by its ability to provide crucial insights into the distribution characteristics of point cloud data. Analyzing the

skewness feature facilitates understanding the skewness patterns exhibited by the point cloud data along different dimensions, i.e., whether the data values are skewed towards the left or right. This enables the identification of inherent asymmetry or skewness phenomena present within the data.

- **Kurtosis Features** [250] In the domain of 3D point cloud data analysis, the kurtosis feature serves as a statistical measure for describing the peakedness and shape of the data distribution. It quantifies the sharpness and peakedness of the point cloud data distribution. Given a point cloud P consisting of N points, where the x, y, and z coordinates of each point are denoted as $X = [x_1, x_2, \ldots, x_n]$, $Y = [y_1, y_2, \ldots, y_n]$, and $Z = [z_1, z_2, \ldots, z_n]$, Formula 6.12 illustrates the computation process for the kurtosis feature vector of the point cloud P.

$$Kurtosis(P) = [kurtosis(X), kurtosis(Y), kurtosis(Z)] \tag{6.12}$$

where $kurtosis(X)$ represents the kurtosis of $X = [x_1, x_2, \ldots, x_n]$, $kurtosis(Y)$ denotes the kurtosis of $Y = [y_1, y_2, \ldots, y_n]$, and $kurtosis(Z)$ signifies the kurtosis of $Z = [z_1, z_2, \ldots, z_n]$. Formula 6.13 outlines the computation process for the kurtosis of the X-coordinates. The procedures for calculating the kurtosis of the Y and Z coordinates follow a similar approach.

$$kurtosis(X) = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{x_i - \mu}{\sigma} \right)^4 - 3 \tag{6.13}$$

where μ represents the mean of $X = [x_1, x_2, \ldots, x_n]$, and σ denotes the standard deviation of X. N denotes the total number of points in P, and x_i represents the X-coordinates of the i-th point in P.

The utilization of the kurtosis feature stems from its ability to provide crucial insights into the distribution characteristics of point cloud data. By analyzing the kurtosis feature, it becomes possible to ascertain the peakedness of data values across different dimensions, thereby discerning the steepness of their distribution.

6.3.3 Mutation Feature Generation

Given a point cloud test set denoted as T and a DNN model denoted as M, we employ the following approach to mutate T and generate mutation features. It is important to note that each test sample is a point cloud composed of thousands of points. Figure 6.1 visually presents one example of point cloud.

- **Mutation generation** Initially, for each test sample (a point cloud) in the test set T, a group of points are randomly selected, and their coordinates are randomly perturbed to generate a mutated point cloud (also called a mutant). This process is repeated N times, each execution being independent and random, resulting in N mutants generated for each test $t \in T$.
- **Mutation feature generation** For $t \in T$, the previous step yields a set of mutants for it, denoted as $\{t'_1, t'_2, \ldots, t'_N\}$. PCPrior compares the predictions made by model M for the test t and each of its mutants t'_i, thereby constructing a mutation feature vector specific to test t. Specifically, if model M produces different predictions for test t and the mutant t'_i, PCPrior sets the i_{th} element of t's mutation feature vector to 1; otherwise, it is set to 0. PCPrior constructs a mutation feature vector for each $t \in T$. Given a test input t (a point cloud), Formula 6.14 describes the above mutation feature (MF) generation process in

PCPrior.

$$V_{MF}[k] = \begin{cases} 1 & \text{if } M(t_k) \neq M(t) \\ 0 & \text{if } M(t_k) = M(t) \end{cases} \tag{6.14}$$

where V_{MF} represents the mutation feature vector generated for the test input t. $V_{MF}[k]$ denotes the k-th value of this feature vector. $M(t_k)$ represents the prediction of the 3D shape classification model M for mutant t_k, and $M(t)$ represents the prediction for the original test input t.

The principle behind leveraging the mutation features of test inputs for test prioritization is that: *A test input t is considered more likely to be misclassified if the evaluated model's predictions for many mutants of t differ from the prediction for t.* This principle draws inspiration from mutation testing techniques employed in traditional software engineering [56, 57]. Besides, our mutation feature generation approach offers several advantages:

❶ **Capturing Model Sensitivity.** The mutation feature generation approach allows to capture the sensitivity of model M to perturbations in the test input. By comparing the predictions of model M for the original test t and its corresponding mutants t'_i, we can identify test instances where even small changes in the input result in different model predictions. Such instances are considered more likely to be misclassified by the DNN model.

❷ **Fine-Grained Analysis.** By constructing a mutation feature vector specific to each test $t \in T$, we obtain a fine-grained analysis of the model's behavior for individual test cases. The mutation feature vector captures the differences between the original test and each of its mutants.

❸ **Interpretability.** The mutation feature vectors provide an interpretable representation of the model's behavior. Each element of the vector indicates whether the evaluated model's result for a specific mutant differs from its result for the original test.

6.3.4 Prediction Feature Generation

The Prediction Feature (PF) captures the probability information of a given test sample belonging to each class. To obtain the Prediction Feature (PF) for a test $t \in T$, initially, we input this test into the target prediction model M. This model is the 3D shape classification model that we evaluated. The model outputs a vector for the test t, denoted as $\{p_1, p_2, \ldots, p_n\}$, where this vector represents the probabilities of test t belonging to each category. Here, p_i denotes the model's prediction probability for test t belonging to category i. For instance, a feature vector [0.1, 0.1, 0.8] signifies that, according to the predictions made by model M, the test input t has a 10% probability of belonging to the first class, a 10% probability of belonging to the second class, and an 80% probability of belonging to the third class. The utilization of Prediction Features has been observed in several prior studies focusing on DNN test optimization, such as Li *et al.*[48] and Feng *et al.*[6].

Given 3D shape classification model M and a test input t, the prediction feature vector of t is obtained based on Formula 6.15.

$$V_{PF}(t) = M(t) = \langle p_{t,1}, p_{t,2}, \cdots, p_{t,C} \rangle \tag{6.15}$$

where $M(t)$ denotes the prediction probability vector of model M for the test t. $p_{t,i}$ represents the probability predicted by model M that the test input t belongs to the i-th category. C signifies the total number of predicted categories by the model M.

6.3.5 Uncertainty Feature Generation

The Uncertainty Features (UF) capture the model's confidence associated with its classification results for each test input $t \in T$. To obtain the UF, we employ six widely used confidence-based metrics [9, 6, 88], namely DeepGini, Vanilla SM, PCS, Entropy, Margin, and Least Confidence. These metrics are selected due to their extensive adoption in quantifying uncertainty in DNN classification tasks and their demonstrated effectiveness [101, 9]. The process of constructing the uncertainty feature vector for each test input $t \in T$ is as follows:

- **Confidence score calculation** We calculate the confidence scores for each test input t using the aforementioned six confidence-based metrics.
- **Feature generation** The uncertainty feature vector is generated by concatenating the obtained confidence scores from the six metrics. Consequently, for each test $t \in T$, a feature vector $[S_1, S_2, S_3, S_4, S_5, S_6]$ is built, where each element S_i represents the confidence score calculated by the i_{th} confidence-based metric for the test input t.

For a given test input t, Formula 6.16 outlines the process of generating its uncertainty features (UF). In Formula 6.16, DeepGini(t) denotes the uncertainty score calculated by DeepGini [6] specifically for the test t, whereas the remaining terms represent uncertainty scores computed by other metrics for measuring uncertainty.

$$V_{UF}(t) = Concat(DeepGini(t), Margin(t), Entropy(t), LC(t), Vanilla(t), PCS(t)) \tag{6.16}$$

6.3.6 Feature Concatenation

For each test input $t \in T$, we integrate four distinct types of features, namely Spatial Features (SF), Mutation Features (MF), Prediction Features (PF), and Uncertainty Features (UF), to construct a final representative feature vector. This feature vector encompasses the relevant information extracted from all feature types associated with the given test input. Subsequently, the constructed feature vector is fed into the ranking models, which are designed to evaluate the likelihood of misclassification for the test input based on its final feature vector. In the subsequent section, we provide a detailed exposition of the methodology employed in the ranking model.

6.3.7 Learning-to-rank

In this step, we employ the LightGBM ranking model [89] to leverage the feature vector of a given test instance $t \in T$ in order to predict its misclassification score. LightGBM is a well-regarded machine learning algorithm based on the Gradient Boosting Decision Tree (GBDT) methodology, renowned for its effectiveness and accuracy. However, due to the binary nature of LightGBM's output, which is not aligned with our objective of estimating the probability of misclassification for a test input, we introduce certain modifications to the original LightGBM algorithm. More specifically, rather than obtaining a binary classification output from the ranking models, which indicates whether the test will be predicted incorrectly, we extract the intermediary output. This intermediate result conveys valuable information regarding the probability of misclassification for each test input.

Upon completion of the training phase (described in Section 6.3.8) of LightGBM, when a feature vector of a test instance is provided as input to the ranking model, we

extract an intermediate value predicted by LightGBM. In the following, we provide a detailed explanation of how the intermediate value is obtained from LightGBM. Initially, the original LightGBM was a binary classification model. For a given test, it can categorize the test into two classes based on its final feature vector (obtained from the above steps), where an output of 0 indicates that the test will be correctly predicted by the model, and an output of 1 indicates that the test will be incorrectly predicted. Its internal logic operates as follows: For a test t_i, LightGBM first generates an **intermediate value**, which signifies the probability of the test being incorrectly predicted by the model. If this intermediate value exceeds 0.5, LightGBM will classify it as 1, indicating that the test is likely to be misclassified by the model. Conversely, if the value is below 0.5, it will be classified as 0, suggesting that the test is likely to be correctly predicted. In PCPrior, rather than letting LightGBM carry out classification, we directly extract this intermediate value for the purpose of test prioritization. Tests with higher intermediate values are considered more likely to be incorrectly predicted and are, therefore, assigned a higher priority.

Specifically, when PCPrior is used for test prioritization, the detailed process of learning-to-rank is as follows: For each test $t_i \in T$, based on its final feature vector, the LightGBM model generates an intermediate value for it, which we denote as F_i, representing the probability of test t_i being predicted incorrectly by the model M. It ranges from 0 to 1. If F_i for a test is closer to 1, it indicates that the test is more likely to be predicted incorrectly by the model. PCPrior ranks all the tests in the test set based on their F_i value. Tests with higher F_i will be prioritized higher.

In the following, we explain the reasons for choosing the LightGBM model as the default model for PCPrior:

- **Improved effectiveness**. In RQ2 (cf. Section 6.5.2), we evaluated the effectiveness of different ranking models on test prioritization. We found that LightGBM and XGBoost performed best across all subjects. However, in most cases of our experiments, LightGBM outperformed XGBoost.
- **Faster training speed**. Moreover, prior work [89] has indicated that LightGBM trains faster than XGBoost. Therefore, compared to XGBoost, LightGBM is more efficient.

6.3.8 Usage of PCPrior

Through the utilization of ranking models, the PCPrior framework is able to predict a misclassification score for each test input within a given test set. These predicted scores are then employed for test prioritization, prioritizing test inputs with higher scores. Specifically, the ranking models undergo pre-training prior to the execution of PCPrior. The training process is presented as follows:

❶ **Training Set Construction:** Given a DNN model M with a point cloud dataset D, the dataset D is initially split into two partitions: the training set R and the test set T, following a 7:3 ratio [92]. The test set remains untouched to evaluate the performance of PCPrior. Based on the training set R, our objective is to build a training set R' for training the ranking models. Firstly, we generate four types of features for each training input $r_i \in R$, using the procedures described in Sections 6.3.2 to 6.3.5. Then, we obtain the final feature vector V_i for each training input r_i, following the guidelines in Section 6.3.6. This final feature vector is utilized to construct the training set R', which serves as the training data for the ranking models. Secondly, we input each training input $r_i \in R$ into the

original model and obtain its classification results, denoted as L_i. By comparing L_i with the ground truth of r_i, we determine whether r_i is misclassified by the model M. If r_i is misclassified, it is labeled as 1; otherwise, it is labeled as 0. This process enables the label construction of the ranking model training set R'.

❷ **Ranking Model Training:** Using the training set R', we proceed to train the ranking models. Upon completion of the training process, the ranking model is capable of producing a misclassification score for a given input, based on the feature vector generated by PCPrior.

6.4 Study design

In this section, we provide a comprehensive exposition of the details pertaining to our study design. Specifically, Section 6.4.1 elucidates the research questions that served as the guiding framework for our investigation. Within Sections 6.4.2 and 6.4.3, we meticulously present the point cloud subjects and measurement metrics that were employed to assess the effectiveness of PCPrior. Furthermore, Section 6.4.4 showcases the five DNN test prioritization methods that were employed as comparative approaches against PCPrior. In Section 6.4.5, we elucidate the design and characteristics of PCPrior variants. Additionally, Section 6.4.6 exhibits the implementation and configuration setup that were utilized in our study.

6.4.1 Research Questions

Our experimental evaluation answers the research questions below.

- **RQ1: How does PCPrior perform in prioritizing test inputs for 3D point clouds?**
  In contrast to existing test prioritization methodologies, our proposed approach, PCPrior, leverages the unique characteristics of point clouds for test prioritization. In this research question, we evaluate the effectiveness of PCPrior by comparing it with existing test prioritization approaches that have been demonstrated effective in prior studies [6, 9] and random selection (baseline).
- **RQ2: How do different ranking models affect the effectiveness of PCPrior?**
  In the original implementation of PCPrior, the LightGBM ranking algorithm [89] was employed to leverage the generated features of test inputs for test prioritization. In this research question, we explore the utilization of alternative ranking algorithms, namely Logistic Regression [251], XGBoost [13], and Random Forest [90], with the objective of examining the influence of ranking models on the effectiveness of PCPrior. To this end, we design a set of variants for PCPrior, each incorporating one of the aforementioned ranking models, while maintaining consistency with the remaining workflow.
- **RQ3: How does the selection of main parameters of PCPrior affect its effectiveness?**
  We conducted an in-depth investigation of the main parameters in PCPrior, with the aim of evaluating whether PCPrior can consistently outperform the compared test prioritization approaches when these parameters undergo modifications.
- **RQ4: How does PCPrior and its variants perform on noisy 3D point clouds?**
  In addition to assessing PCPrior and its variants on natural datasets, we undertake an evaluation that encompasses noisy 3D point clouds, thereby facilitating an

in-depth examination of their effectiveness.

- **RQ5: To what extent does each type of features contribute to the effectiveness of PCPrior?**
  In PCPrior, we generate four different types of features from each test input for test prioritization, namely Spatial Features, Mutation Features, Prediction Features and Uncertainty Features, as elaborated in Section 6.3. In this research question, we compare the contributions of different types of features on the effectiveness of PCPrior.
- **RQ6: Can PCPrior and uncertainty-based methods be employed to guide the retraining process for enhancing a 3D shape classification model?**
  Faced with a substantial volume of unlabeled inputs and a constrained time budget, manually labeling all inputs for retraining a 3D shape classification model becomes impractical. Active learning is acknowledged as a practical solution for reducing data labeling costs [252]. This approach focuses on selecting an informative subset of samples to retrain the model, aiming to improve model performance with minimal labeling costs. In this research question, we investigate the effectiveness of PCPrior and uncertainty-based metrics in selecting informative retraining inputs to improve the performance of 3D shape classification models.

6.4.2 Models and Datasets

The effectiveness of PCPrior and the compared test prioritization approaches [6, 9] was evaluated using a set of 165 subjects. Essential details regarding these subjects are presented in Table 6.1, which highlights the matching relationship between the point cloud dataset and the DNN models. In particular, the "#Size" column indicates the size of the dataset, while the "Type" column denotes the type of the dataset, with "Original" representing natural data and "Noisy" indicating noisy data.

Among the 165 subjects, 15 subjects (3 point cloud datasets $\times$ 5 models) were generated using natural datasets, while the remaining 150 subjects were generated using noisy datasets. To generate a noisy dataset from the original test set T, each test instance $t \in T$ undergoes a modification. Specifically, within each test instance t (a point cloud), approximately 30% of the points undergo a random offset, while the remaining 70% of the points remain unchanged. The 30% ratio is derived from the reasonable range of noise injection proportions provided in the existing work [253]. The 150 subjects derived from noisy data were obtained as follows: For each original dataset, we generated 10 noisy datasets, resulting in a total of 30 noisy datasets. Each noisy dataset was paired with five different models, resulting in a total of 150 subjects (30 datasets $\times$ 5 models).

In the following part, we present the description of the 3D point cloud datasets and DNN models utilized in our study.

6.4.2.1 Datasets

In our research, we employed three prominent point cloud datasets, namely ModelNet40 [39], ShapeNet [40], and S3DIS [41]. These datasets are widely adopted within the academic community and have consistently served as benchmarks for several state-of-the-art point cloud studies [254, 255, 256].

- **ModelNet40** [39]: ModelNet40 consists of 12,311 point clouds in 40 categories (e.g., airplane, car, plant, lamp). It encompasses synthetic object point clouds

Table 6.1: 3D Point cloud datasets and models

ID	Dataset	# Size	Model	Type
1	ModelNet	12311	DGCNN	Original, Noisy
2	ModelNet	12311	PointConv	Original, Noisy
3	ModelNet	12311	MSG	Original, Noisy
4	ModelNet	12311	SSG	Original, Noisy
5	ModelNet	12311	PointNet	Original, Noisy
6	S3DIS	9813	DGCNN	Original, Noisy
7	S3DIS	9813	PointConv	Original, Noisy
8	S3DIS	9813	MSG	Original, Noisy
9	S3DIS	9813	SSG	Original, Noisy
10	S3DIS	9813	PointNet	Original, Noisy
11	ShapeNet	53107	DGCNN	Original, Noisy
12	ShapeNet	53107	PointConv	Original, Noisy
13	ShapeNet	53107	MSG	Original, Noisy
14	ShapeNet	53107	SSG	Original, Noisy
15	ShapeNet	53107	PointNet	Original, Noisy

and stands as a paramount benchmark for point cloud analysis. Renowned for its diverse range of categories, meticulous geometric shapes, and methodical dataset construction, ModelNet40 has garnered significant popularity in the research community [254].

- **ShapeNet** [40]: ShapeNet dataset is a widely recognized and extensively used benchmark in the field of 3D shape classification. The ShapeNet dataset utilized in our study consists of 50 categories and a total of 53,107 samples. These categories include chairs, tables, cars, airplanes, animals, etc.
- **Stanford Large-Scale 3D Indoor Spaces Dataset (S3DIS)** [41]: The S3DIS dataset is widely recognized for its comprehensive representation of diverse indoor environments, encompassing various real-world scenes encountered in indoor settings. The S3DIS dataset utilized in our study consists of 9,813 samples, classified into 13 categories (e.g., office, meeting room, and open space).

6.4.2.2 Models

- **PointConv** [17]: PointConv is a convolutional neural network operator specifically designed for processing 3D point clouds characterized by non-uniform sampling. By training multi-layer perceptrons using local point coordinates, PointConv approximates continuous weight and density functions within convolutional filters. In this way, deep convolutional networks can be directly constructed on 3D point clouds, enabling efficient and effective analysis and processing.
- **Dynamic Graph Convolutional Neural Network (DGCNN)** [16]: DGCNN is a deep learning architecture specifically designed for processing and analyzing 3D point cloud data. The key idea behind DGCNN is to exploit the intrinsic spatial relationships present in point clouds by modeling them as graphs. By leveraging graph convolutions and dynamically adapting the graph structure based on the input data, DGCNN can effectively learn and process point cloud representations, making it suitable for point cloud classification tasks.
- **PointNet** [38]: PointNet is a widely-adopted deep learning architecture specifically tailored for 3D point cloud data. The architecture includes a shared multi-layer

perceptron (MLP) with max-pooling to extract local features from individual points and a symmetric function to aggregate the global features across all points. By employing T-Net layers, PointNet is able to learn transformation matrices that aid in aligning and transforming input point clouds, enhancing the model's robustness to input variations. PointNet has demonstrated impressive capabilities in 3D shape classification tasks, establishing it as an effective approach for point cloud analysis.

- **MSG** [42]: MSG refers to multi-scale grouping. The MSG approach involves sampling representative points and grouping nearby points within a specified radius. This allows for the extraction of local features at multiple scales, enabling hierarchical feature learning from point sets.
- **SSG** [42]: SSG, an acronym for Single-Scale Grouping, denotes a simplified variant of the multi-scale grouping architecture. The essence of SSG lies in the partitioning of a point cloud into local regions of fixed size while disregarding the consideration of multiple scales. Within each region, a representative subset of points is judiciously sampled, and proximate points falling within a predefined radius are grouped together. This approach facilitates local feature extraction while avoiding the intricate intricacies associated with handling diverse scales.

6.4.3 Measurements

The goal of PCPrior is to prioritize the possibly-misclassified test inputs in the context of 3D point cloud data. Thus following the existing work [6], we adopted Average Percentage of Fault-Detection (APFD) and Percentage of Fault Detected (PFD) to measure the effectiveness of PCPrior, the compared approaches, and the variants of PCPrior.

- **Average Percentage of Fault-Detection (APFD)** APFD [11] is a widely recognized metric for assessing the effectiveness of prioritization techniques. A higher APFD value indicates a quicker rate of detecting misclassifications. The calculation of APFD values is based on Formula 6.17.

$$APFD = 1 - \frac{\sum_{i=1}^{k} o_i}{kn} + \frac{1}{2n} \tag{6.17}$$

  where n denotes the total number of test inputs, and the variable k represents the number of test inputs in T that will be incorrectly predicted by the model. The index o_i pertains to the position of the i_{th} misclassified test within the prioritized test set. Specifically, o_i represents an integer value indicating the position of the i_{th} misclassified test within the prioritized test set.
  Based on the existing study [6], we normalize the APFD values to [0,1]. A prioritization approach is considered better when the APFD value is closer to 1. This is because: a larger APFD value corresponds to a smaller value of $\sum_{i=1}^{k} o_i$. Here, $\sum_{i=1}^{k} o_i$ represents the total index sum of misclassified tests within the prioritized list. A smaller $\sum_{i=1}^{k} o_i$ implies that the evaluated test prioritization method assigns higher priority to misclassified tests, positioning them at the front of the ranked test set. This effective detection of misclassified tests demonstrates the efficacy of the test prioritization approach. Therefore, a larger APFD value serves as an indicator of better effectiveness for test prioritization strategies.
- **Percentage of Fault Detected (PFD)** PFD refers to the proportion of detected misclassified test inputs among all misclassified tests. Higher PFD values indicate

better test prioritization effectiveness. PFD is calculated based on Formula 6.18.

$$PFD = \frac{\#N_d}{\#N} \tag{6.18}$$

where $\#N_d$ is the number of misclassified test inputs that have been detected. $\#N$ denotes the total number of misclassified tests. In our study, we evaluated the PFD of PCPior and the compared test prioritization approaches against different ratios of prioritized tests. We utilize **PFD-n** to denote the first n% prioritized test inputs.

6.4.4 Compared Approaches

This study employed five comparative approaches, which included a baseline approach (random selection) and four DNN test prioritization techniques. The selection of these methods was driven by multiple factors: 1) These approaches can be adapted for test prioritization in the context of 3D point cloud data; 2) These approaches were proposed within the DL testing community and have been previously demonstrated as effective for DNNs; 3) These approaches provide open-source implementations.

- **Random selection** [102] Random selection is the baseline in our study. Random selection involves the randomized determination of the execution order for test inputs. This means that the sequencing of test inputs is established in a completely arbitrary manner, devoid of any predetermined patterns or logical arrangements.
- **DeepGini** [6] DeepGini utilizes the Gini coefficient, which is a statistical metric used to assess the probability of misclassification, in order to facilitate the ranking of test inputs. The Gini score is calculated according to Formula 6.19, which is presented below:

$$G(t) = 1 - \sum_{i=1}^{N} (p_i(t))^2 \tag{6.19}$$

where $G(t)$ represents the probability of the test input t being misclassified. $p_i(t)$ denotes the probability that the test input t is predicted to belong to label i. N represents the total amount of categories that the input can be assigned to.

- **Prediction-Confidence Score (PCS)** PCS [9] assigns rankings to test inputs based on the difference between the predicted class and the second most confident class in the softmax likelihood. A smaller difference indicates that the model is less certain about the prediction for a particular test input. These uncertain tests are given higher priority and are placed at the front of the test set. The calculation of this difference is defined by Formula 6.20 as follows:

$$P(x) = l_k(x) - l_j(x) \tag{6.20}$$

where $l_k(x)$ refers to the most confident prediction probability. $l_j(x)$ refers to the second most confident prediction probability.

- **Vanilla Softmax** [9] Vanilla Softmax measures the difference between the maximum activation probability in the output softmax layer and the ideal value of 1 for each test input. This disparity reflects the degree of uncertainty associated with the model's predictions. Test inputs with larger disparities are considered more likely to be misclassified by the model. The specific computation of this disparity

is illustrated by Formula 6.21, which provides a clear and concise representation of the underlying mathematical calculations.

$$\mathrm{V}(x) = 1 - \max_{c=1}^{C} l_c(x) \tag{6.21}$$

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

- **Entropy** [9] Entropy serves as a criterion for ranking test inputs based on the entropy of their softmax likelihood. Higher entropy values indicate greater uncertainty in the model's predictions for those inputs. Consequently, test inputs with higher entropy are considered more likely to be misclassified by the model. As a result, they are given higher priority and placed at the beginning of the test set.

6.4.5 Variants of PCPrior

We conducted an investigation into the influence of different ranking models on the effectiveness of PCPrior. To this end, we proposed five variants of PCPrior, namely PCPrior^L, PCPrior^X, PCPrior^R, PCPrior^D, and PCPrior^T, which utilize Logistic Regression [251], XGBoost [13], Random Forest [90], DNNs [257], and TabNet [258] as the ranking model, respectively. It is essential to emphasize that apart from the variation in ranking models, the execution workflow of these derived variants remains identical to that of the original PCPrior approach.

Furthermore, we extended the modifications applied to the LightGBM ranking model of PCPrior to the ranking models employed by the variants of PCPrior. Specifically, instead of making the ranking models provide a binary classification output (i.e., indicating whether the test will be predicted incorrectly by the model), we extract the intermediate output, which can indicate the probability of misclassification for each test input. Consequently, we obtain a misclassification score for each test input, which can be effectively utilized for test prioritization. In the following sections, we provide a comprehensive explanation of the specific ranking models utilized in each variant of PCPrior.

- **PCPrior^L:** In the context of PCPrior^L, we employ the Logistic Regression algorithm [91] as the ranking model. Logistic Regression is a statistical modeling technique that employs a logistic function to establish the relationship between a categorical dependent variable and one or more independent variables.
- **PCPrior^X:** In the context of PCPrior^X, we utilize the XGBoost ranking algorithm [13] to estimate the misclassification score of a test input based on its corresponding feature vector. XGBoost is a powerful gradient-boosting technique that integrates decision trees to enhance prediction accuracy. By leveraging its ensemble learning capabilities, XGBoost effectively captures complex relationships within the data, enabling accurate estimation of the likelihood of misclassification for each test input.
- **PCPrior^R:** In the context of PCPrior^R, we employ the Random Forest algorithm [90] as the ranking model. Random Forest is an ensemble learning algorithm that constructs multiple decision trees. The predictions from individual trees are combined using averaging or voting mechanisms to produce the final prediction. Random Forest is known for its ability to handle high-dimensional data and capture intricate interactions among features. By leveraging these strengths,

PCPrior^R accurately estimates the misclassification score for each test input, aiding in effective test prioritization.

- **PCPrior^D:** In the context of PCPrior^D, we utilize a DNN model as the ranking model, derived from a prior investigation [257]. This DNN model is capable of producing a misclassification score for a given test input, relying on its feature vector generated by PCPrior.
- **PCPrior^T:** In the context of PCPrior^T, we utilize TabNet [258] as the ranking model. TabNet is a DNN architecture specifically designed for tabular data. It has been demonstrated to be more effective than XGBoost and LightGBM in a previous study [258].

6.4.6 Implementation and Configuration

We implemented PCPrior in Python, utilizing the PyTorch 2.0.0 framework [103]. To enable comparison with other approaches, we integrated existing implementations of the compared methods [9, 6] into our experimental pipeline, specifically tailored for test prioritization of 3D point cloud data. To generate mutation features, we created 30 mutants for each test sample. Regarding the configuration of the ranking models employed in PCPrior, we utilized XGBoost 1.7.4, LightGBM 3.3.5, and scikit-learn 1.0.2 frameworks. Furthermore, we made specific parameter selections: for LightGBM, the learning rate was set to 0.1; for Logistic Regression, the parameter *max_iter* was set to 100; for XGBoost, the learning rate was set to 0.3; and for the random forest algorithm, the number of estimators was set to 100. 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 Big Sur 11.6, equipped with an Intel Core i9 CPU and 64 GB of RAM. In total, we conducted experiments on 165 subjects, consisting of 15 subjects based on natural inputs and 150 subjects based on noisy inputs.

6.5 Results and analysis

6.5.1 RQ1: Performance of PCPrior

Objectives: We investigate the effectiveness and efficiency of PCPrior, comparing it with several existing test prioritization approaches.

Experimental design: We conducted experiments to evaluate the performance of PCPrior from the following three aspects.

- **Effectiveness evaluation on natural datasets.** We employed a set of 15 subjects constructed from 3D point cloud datasets to evaluate the effectiveness of PCPrior. Table 3.1 presents the basic information of these subjects. In order to assess the performance of PCPrior, we carefully selected four test prioritization approaches, namely DeepGini, Vanilla SM, PCS, and entropy, alongside a baseline method (i.e., random selection), for comparative analysis. Moreover, we utilized two measurement metrics, specifically the Average Percentage of Fault-Detection (APFD) and the Percentage of Fault Detected (PFD), to evaluate the effectiveness of PCPrior and the compared approaches. A detailed explanation of the calculations for these metrics can be found in Section 6.4.3.
- **Statistical analysis.** Due to the inherent randomness in the model training process, we performed statistical analysis by conducting the experiments ten times. Specifically, for each subject, which refers to a point cloud dataset paired with a

DNN model, we generated ten distinct models through separate training processes. The average results are reported in our experimental findings. Moreover, for each subject, we calculated the variance of ten repeated experimental results for each test prioritization method to demonstrate the stability of PCPrior better.
To further validate the stability and reliability of the experimental findings, we calculated p-values associated with the results. Specifically, we employed the **paired two-sample t-test** [104] to calculate the p-value, a commonly used statistical method for evaluating differences between two related data sets. The essential steps involved are: 1) selecting two related sets of data, 2) computing the difference for each corresponding pair of data points, and 3) analyzing these differences to ascertain if there is a statistically significant disparity between the two data sets. In the paired two-sample t-test approach, the significance of the results is determined by the p-value. Generally, if the p-value is less than 10^{-05}, it is considered that the difference between the two sets of data is statistically significant [105]. Additionally, we quantify the magnitude of the difference between the two sets of results through the *Effect Size*. Specifically, we use Cohen's d for measuring the effect size [106]. Wherein, $|d| < 0.2$ – "negligible," $|d| < 0.5$ – "small," $|d| < 0.8$ – "medium," otherwise – "large". To ensure that the difference between the results of PCPrior and the compared approach is "non-negligible", we require that the value of d is greater than or equal to 0.2.

- **Efficiency evaluation.** In addition to evaluating the effectiveness of PCPrior, we conducted an assessment of its efficiency and compared it with the selected test prioritization methods. Specifically, we quantified the time required for each step of PCPrior to measure its efficiency. By analyzing the execution time of PCPrior, we aim to gain insights into its computational efficiency and its potential for practical application in real-world scenarios.

Results: The experimental findings pertaining to Research Question 1 (RQ1) are presented in Table 6.2, Table 6.3, Table 6.4, Table 6.5, Table 6.6, Table 6.7 and Figure 6.3. Table 6.2 and Table 6.3 offer a comparative analysis, employing the APFD metric, between PCPrior and the comparative methods. Conversely, Table 6.6 and Figure 6.3 provide an assessment of effectiveness using the PFD metric. It is important to note that we highlight the approach with the highest effectiveness for each case in grey. Additionally, Table 6.7 offers a comparison of the efficiency between PCPrior and the evaluated test prioritization approaches.

Notably, Table 6.2 reveals that across all 15 subjects, PCPrior consistently outperforms all comparative methods in terms of its APFD. Specifically, the range of APFD values for PCPrior spans from 0.781 to 0.905, while the range for the comparative methods lies between 0.495 and 0.853. Moreover, Table 6.3 further highlights the average APFD value for PCPrior and its relative improvement compared to the comparative methods. We see that PCPrior achieves an average APFD of 0.836, whereas the average APFD of the comparative methods falls within the range of 0.501 to 0.754. The improvement observed in PCPrior, relative to the comparative methods, ranges from 10.99% to 66.94%. These findings demonstrate that PCPrior performs better than all the comparative test prioritization methods in terms of the APFD metric.

The comparative analysis presented in Table 6.6 employs the PFD metric to exhibit the comparison between PCPrior and various DNN test prioritization methods. Notably, from prioritizing 10% to 70% of the dataset, PCPrior consistently

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

Approach	ModelNet					S3DIS					ShapeNet				
	DGCNN	PointConv	MSG	SSG	PointNet	DGCNN	PointConv	MSG	SSG	PointNet	DGCNN	PointConv	MSG	SSG	PointNet
Random	0.503	0.503	0.489	0.482	0.509	0.514	0.493	0.505	0.509	0.504	0.502	0.495	0.496	0.501	0.511
DeepGini	0.763	0.719	0.757	0.748	0.734	0.715	0.700	0.699	0.703	0.653	0.846	0.740	0.824	0.837	0.793
VanillaSM	0.768	0.725	0.763	0.754	0.737	0.718	0.702	0.702	0.705	0.657	0.850	0.745	0.827	0.839	0.797
PCS	0.770	0.729	0.767	0.756	0.737	0.717	0.699	0.700	0.702	0.657	0.853	0.746	0.830	0.841	0.800
Entropy	0.751	0.707	0.743	0.735	0.724	0.703	0.692	0.690	0.696	0.644	0.831	0.728	0.813	0.829	0.780
PCPrior	0.807	0.781	0.809	0.796	0.793	0.833	0.827	0.815	0.820	0.817	0.905	0.852	0.897	0.904	0.891

Table 6.3: Effectiveness improvement of PCPrior over the compared approaches in terms of the APFD values on natural datasets

Approach	# Best cases	Average APFD	Improvement(%)
Random	0	0.501	66.94
DeepGini	0	0.749	11.72
VanillaSM	0	0.753	11.14
PCS	0	0.754	10.99
Entropy	0	0.738	13.38
PCPrior	15	0.836	-

outperforms all comparative methods in terms of PFD. To facilitate a more intuitive comparison, Figure 6.3 showcases two line graphs with PFD as the y-axis, illustrating the cases of ModelNet dataset with DGCNN model and ShapeNet dataset with PointNet model, respectively. All the results can be found on our Github[2]. In the figures, PCPrior is depicted by the red lines, while the baseline is represented by the pink lines. Visual analysis reveals that PCPrior consistently achieves a higher PFD value when contrasted with DeepGini, entropy, Vanilla SM, PCS, and random methods. These experimental results demonstrate that PCPrior outperforms all comparative test prioritization methods in terms of the PFD metric.

As stated in the experimental design, a statistical analysis was conducted to ensure the stability of our findings. To this end, all experiments were repeated ten times for each subject. The statistical analysis reveals a p-value lower than 10^{-05}, providing strong evidence that PCPrior consistently outperforms the compared approaches in the context of test prioritization. Table 6.4 presents detailed results from the statistical analysis. The analysis employs two primary metrics: p-value and effect size. As outlined in the experimental design, a p-value less than 10^{-05} indicates that the difference between two data sets is statistically significant [105]. Furthermore, an effect size ≥ 0.2 suggests that the difference is "non-negligible." In Table 6.4, we observed that all the p-values between PCPrior and the compared approaches consistently fall below 10^{-05}, indicating that PCPrior statistically outperforms all the compared test prioritization methods. For example, the p-value for the difference in experimental results between PCPrior and DeepGini is 2.039×10^{-07}. The p-value between PCPrior and VanillaSM is 4.403×10^{-07}. Additionally, the experimental results for both PCPrior and the compared approaches show effect sizes exceeding 0.2, confirming a non-negligible difference. Moreover, we found that all the effect sizes are even greater than 0.8. For example, the effect size of PCPrior and VanillaSM is 2.273. According to Cohen's d [106], this means that the difference in experimental results between PCPrior and the compared methods is not only statistically significant but also relatively "large" in scale.

Moreover, for each case, we calculated the variance of ten repeated experimental results with respect to each test prioritization method, as presented in Table 6.5. It

[2]https://github.com/yinghuali/PCPrior/tree/main/result

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

	Random	DeepGini	VanillaSM	PCS	Entropy
PCPrior (p-value)	3.444×10^{-14}	2.039×10^{-07}	4.403×10^{-07}	8.663×10^{-07}	3.071×10^{-08}
PCPrior (effect size)	7.854	2.423	2.273	2.148	2.822

Table 6.5: Variance in experimental results ($\times 10^{-3}$) for PCPrior and the compared approaches across ten repetitions

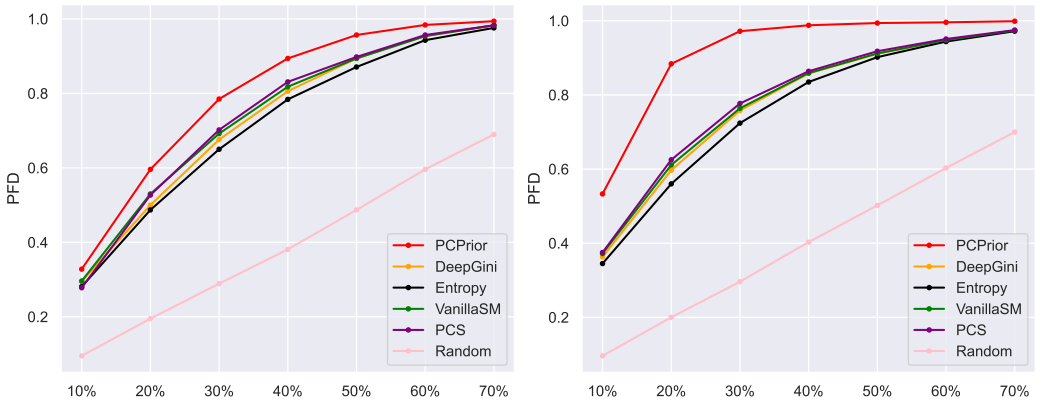
Approach	ModelNet					S3DIS					ShapeNet				
	DGCNN	PointConv	MSG	SSG	PointNet	DGCNN	PointConv	MSG	SSG	PointNet	DGCNN	PointConv	MSG	SSG	PointNet
Random	0.044	0.012	0.056	0.026	0.054	0.079	0.036	0.102	0.089	0.035	0.037	0.008	0.018	0.047	0.021
DeepGini	0.026	0.200	0.050	0.021	0.031	0.071	0.405	0.026	0.045	0.038	0.027	0.064	0.015	0.009	0.035
VanillaSM	0.022	0.196	0.042	0.022	0.025	0.071	0.396	0.027	0.052	0.029	0.025	0.065	0.015	0.010	0.036
PCS	0.013	0.209	0.031	0.024	0.022	0.068	0.430	0.030	0.051	0.024	0.022	0.070	0.015	0.012	0.033
Entropy	0.030	0.199	0.061	0.021	0.039	0.075	0.404	0.019	0.036	0.053	0.035	0.054	0.020	0.008	0.033
PCPrior	0.005	0.112	0.022	0.011	0.042	0.030	0.375	0.014	0.017	0.030	0.008	0.073	0.003	0.001	0.002

Table 6.6: Average comparison results among PCPrior and the compared approaches on natural data in terms of PFD

Data	Approach	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60	PFD-70
ModelNet	Random	0.100	0.206	0.300	0.398	0.498	0.602	0.699
	DeepGini	0.263	0.467	0.641	0.774	0.874	0.935	0.973
	VanillaSM	0.269	0.483	0.658	0.785	0.875	0.936	0.973
	PCS	0.261	0.488	0.664	0.794	0.881	0.938	0.974
	Entropy	0.253	0.452	0.612	0.746	0.851	0.923	0.968
	PCPrior	0.305	0.567	0.760	0.882	0.950	0.983	0.994
S3DIS	Random	0.101	0.202	0.297	0.400	0.499	0.602	0.705
	DeepGini	0.222	0.402	0.554	0.684	0.789	0.873	0.929
	VanillaSM	0.228	0.409	0.563	0.688	0.790	0.874	0.929
	PCS	0.222	0.410	0.556	0.690	0.789	0.875	0.928
	Entropy	0.212	0.391	0.535	0.663	0.775	0.867	0.926
	PCPrior	0.341	0.629	0.829	0.931	0.972	0.989	0.995
ShapeNet	Random	0.099	0.200	0.297	0.395	0.495	0.597	0.694
	DeepGini	0.386	0.632	0.789	0.878	0.928	0.959	0.979
	VanillaSM	0.399	0.647	0.793	0.879	0.928	0.959	0.979
	PCS	0.403	0.656	0.801	0.884	0.932	0.960	0.979
	Entropy	0.368	0.597	0.758	0.860	0.919	0.955	0.977
	PCPrior	0.555	0.865	0.961	0.984	0.992	0.996	0.998

is important to note that the unit for the table is 10^{-3}. For instance, in the second row, the first number, 0.026, represents that for the ModelNet dataset, under the DGCNN model, the variance of ten repeated experimental results for the DeepGini method is 0.026×10^{-3}. The cases highlighted in grey represent the test prioritization method with the minimum variance for each subject. We see that for 66.7% (10 out of 15) of subjects, PCPrior has the smallest variance. Furthermore, the variance range for PCPrior is 0.001×10^{-3} to 0.375×10^{-3}. In contrast, the variance range for comparative methods is 0.008×10^{-3} to 0.430×10^{-3}. The above experimental results indicate that the variance of PCPrior's results is generally lower compared to the comparative test prioritization methods, suggesting that PCPrior is relatively more stable.

Table 6.7 provides a comprehensive comparison of the efficiency between PCPrior and the compared test prioritization approaches. A noteworthy distinction between our proposed method and the comparative approaches pertains to the requirement of training a ranking model and generating features. As can be observed from Table 6.7, the overall time taken by PCPrior is approximately 6 minutes and 32 seconds. Specifically, the average training time for the PCPrior ranking model amounts to 32 seconds, while the average time for feature generation is 6 minutes. The final



a) ModelNet, DGCNN **b)** ShapeNet, PointNet

Figure 6.3: Test prioritization effectiveness among PCPrior and the compared approaches for ModelNet with DGCNN and ShapeNet with PointNet. X-Axis: the percentage of prioritized tests; Y-Axis: the percentage of detected miscalssified tests.

Table 6.7: Time cost of PCPrior and the compared test prioritization approaches

Time cost	Approach					
	PCPrior	Random	DeepGini	VanillaSM	PCS	Entropy
Feature generation	6 min	-	-	-	-	-
Ranking model training	32 s	-	-	-	-	-
Prediction	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s

prediction time of the compared approaches is less than 1s. Although PCPrior is not as efficient as confidence-based test prioritization approaches, the effectiveness improvement of PCPrior relative to confidence-based methods is 10.99%~13.38%. Considering the trade-off between effectiveness and efficiency, PCPrior remains a practical option.

Answer to RQ1: *PCPrior consistently demonstrates better performance compared to all the evaluated test prioritization approaches (i.e., DeepGini, Vanilla SM, PCS, Entropy, and Random) in the field of test prioritization for 3D point cloud data, as assessed by both the APFD and PFD metrics. Specifically, the average improvement achieved in terms of APFD ranges from 10.99% to 66.94%. While PCPrior is not as efficient as confidence-based methods, considering the trade-off between effectiveness and efficiency, it remains a practical option.*

6.5.2 RQ2: Influence of ranking models

Objectives: We investigate the impact of various ranking models on the effectiveness of PCPrior.

Experimental design: We proposed five variants of PCPrior that incorporate different ranking models. In addition to the ranking models, the other procedures of these methods remain identical to PCPrior. The five variants are PCPrior^{L}, PCPrior^{X}, PCPrior^{R}, PCPrior^{D}, and PCPrior^{T}, which utilize Logistic Regression [251], XGBoost [13], Random Forest [90], DNNs [257], and TabNet [258] as the ranking model, respectively. We evaluated the impact of these ranking models on the effectiveness of PCPrior by assessing the performance of these variants on natural datasets utilizing both the APFD and PFD metrics.

Results: The experimental results for Research Question 2 (RQ2) are presented in Table 6.8 and Table 6.9. Table 6.8 showcases the comparison between PCPrior and

Table 6.8: Effectiveness comparison among PCPrior and PCPrior Variants in terms of the APFD values on natural datasets

Approach	ModelNet					S3DIS					ShapeNet				
	DGCNN	PointConv	MSG	SSG	PointNet	DGCNN	PointConv	MSG	SSG	PointNet	DGCNN	PointConv	MSG	SSG	PointNet
PCPrior^{L}	0.792	0.766	0.781	0.766	0.756	0.732	0.709	0.710	0.707	0.666	0.855	0.789	0.841	0.849	0.804
PCPrior^{X}	0.802	0.778	0.804	0.791	0.792	0.832	0.821	0.818	0.817	0.815	0.910	0.856	0.896	0.907	0.892
PCPrior^{R}	0.790	0.765	0.794	0.773	0.769	0.781	0.791	0.758	0.773	0.775	0.883	0.817	0.868	0.878	0.865
PCPrior^{D}	0.793	0.769	0.791	0.779	0.767	0.748	0.744	0.740	0.741	0.723	0.871	0.831	0.871	0.877	0.858
PCPrior^{T}	0.779	0.758	0.765	0.778	0.766	0.739	0.728	0.724	0.724	0.701	0.899	0.850	0.890	0.894	0.885
PCPrior	0.807	0.781	0.809	0.796	0.793	0.833	0.827	0.815	0.820	0.817	0.905	0.852	0.897	0.904	0.891

its variants in terms of the APFD metric, while Table 6.9 presents their comparison based on the PFD metric.

In Table 6.8, we see that PCPrior, which employs LightGBM as the ranking model, performs the best in 66.67% (10 out of 15) of the cases. PCPrior^{X}, which utilizes XGBoost as the ranking model, performs the best in the remaining 33.3% (5 out of 15) cases. Furthermore, Table 6.9 presents a comparison of the effectiveness of PCPrior and its variants from the perspective of the PFD metric. We see that PCPrior performs the best in 61.9% (13 out of 21) cases, while PCPrior^{X} performs the best in 38.1% (8 out of 21) of the cases. The aforementioned experimental results illustrate that the ranking models employed by PCPrior and PCPrior^{X}, specifically LightGBM and XGBoost, can better utilize the generated test input features for test prioritization.

Surprisingly, despite existing studies [258] mentioning that TabNet is more effective than XGBoost and LightGBM in their evaluated datasets when applied to PCPrior for the purpose of test prioritization, the effectiveness of PCPrior (which utilizes the LightGBM model) is higher than that of PCPrior^{T} (which utilize the TabNet model). We can see that, in Table 6.8, PCPrior's APFD ranges from 0.781 to 0.905, while PCPrior^{T}'s APFD ranges from 0.701 to 0.894. This suggests that, compared to TabNet, LightGBM performs better in leveraging the features (generated by PCPrior) for test prioritization. Some potential reasons include: 1) Different datasets and their distributions can impact the training of classification models, thereby affecting their performance; 2) The size of the dataset can also influence the model's performance. The experimental results demonstrate that LightGBM is more suitable and compatible with the feature dataset constructed by PCPrior.

Answer to RQ2: *PCPrior and PCPrior^{X} exhibits better effectiveness in test prioritization compared to other PCPrior variants, thereby suggesting that the ranking model employed by PCPrior and PCPrior^{X}, namely LightGBM, and XGBoost, can better utilize the generated features of test inputs for test prioritization.*

6.5.3 RQ3: Impact of Main Parameters in PCPrior

Objectives: We investigate the impact of main parameters on the effectiveness of PCPrior for test prioritization.

Experimental design: Building upon the parameter selection and consideration of parameter values in previous research [10], we conducted a systematic investigation to analyze the impact of key parameters in PCPrior. Specifically, we focused on three parameters: *max_depth* (representing the maximum tree depth for each LightGBM model), *colsample_bytree* (indicating the sampling ratio of feature columns when constructing each tree), and *learning_rate* (referring to the boosting learning rate) in the LightGBM ranking algorithm. For our investigation, we performed experiments on all subjects within the natural dataset. By observing the performance variations

Table 6.9: Average comparison results among PCPrior and PCPrior Variants in terms of the PFD values on natural datasets

Data	Approach	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60	PFD-70
ModelNet	PCPrior^{L}	0.283	0.515	0.707	0.832	0.913	0.964	0.985
	PCPrior^{X}	0.304	0.554	0.744	0.876	0.949	0.981	0.992
	PCPrior^{R}	0.289	0.530	0.716	0.842	0.920	0.968	0.990
	PCPrior^{D}	0.295	0.541	0.719	0.847	0.927	0.967	0.985
	PCPrior^{T}	0.283	0.524	0.706	0.831	0.903	0.950	0.981
	PCPrior	0.305	0.567	0.760	0.882	0.950	0.983	0.994
S3DIS	PCPrior^{L}	0.226	0.424	0.578	0.712	0.811	0.885	0.932
	PCPrior^{X}	0.335	0.619	0.831	0.934	0.975	0.988	0.996
	PCPrior^{R}	0.285	0.524	0.709	0.841	0.922	0.969	0.990
	PCPrior^{D}	0.262	0.467	0.634	0.770	0.857	0.916	0.959
	PCPrior^{T}	0.243	0.440	0.612	0.743	0.835	0.903	0.952
	PCPrior	0.341	0.629	0.829	0.931	0.972	0.989	0.995
ShapeNet	PCPrior^{L}	0.427	0.690	0.832	0.907	0.944	0.967	0.980
	PCPrior^{X}	0.561	0.871	0.964	0.987	0.993	0.995	0.997
	PCPrior^{R}	0.485	0.767	0.899	0.955	0.981	0.991	0.995
	PCPrior^{D}	0.476	0.776	0.905	0.954	0.975	0.983	0.990
	PCPrior^{T}	0.543	0.843	0.948	0.976	0.986	0.992	0.995
	PCPrior	0.555	0.865	0.961	0.984	0.992	0.996	0.998

of PCPrior as these parameters changed, we aimed to gain insights into the influence of parameters on the effectiveness of PCPrior.

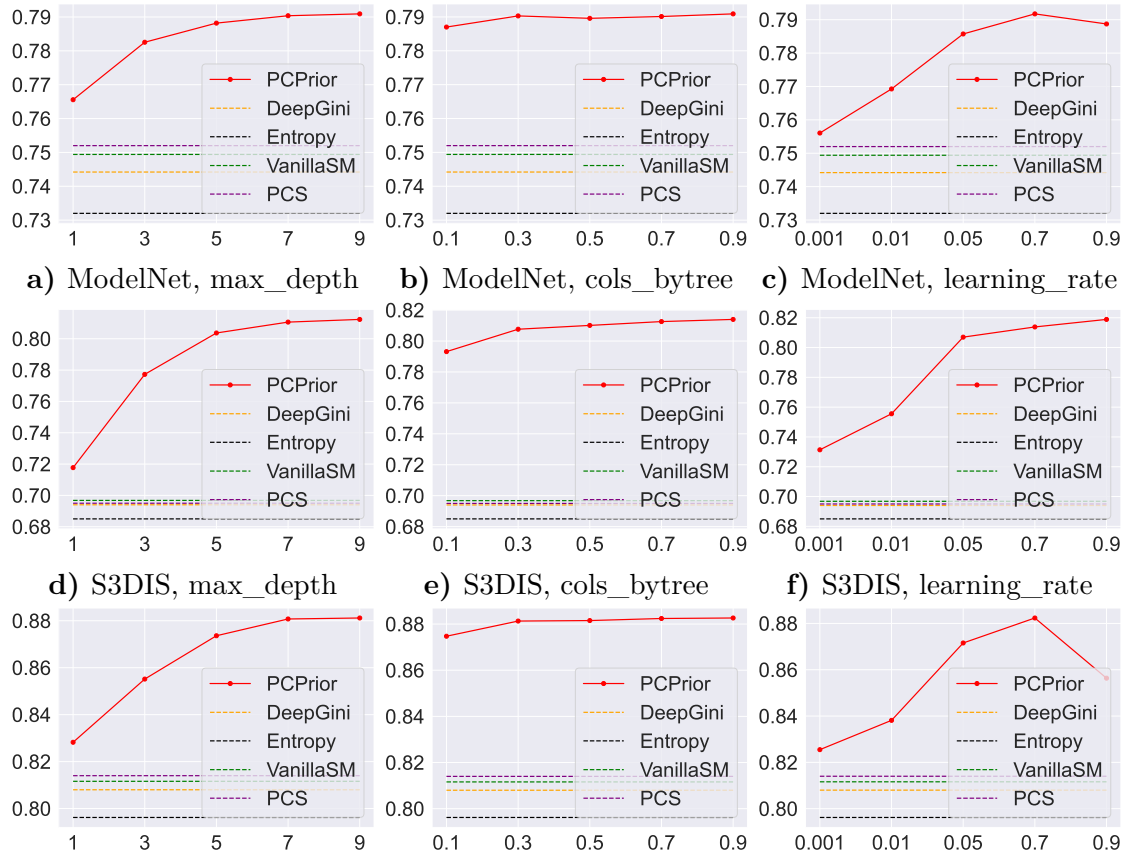
Results: The experimental results of RQ3 are presented in Figure 6.4, showcasing the effectiveness of PCPrior under diverse parameter settings based on average APFD values across the 15 subjects. The solid red line represents PCPrior, while the dashed lines depict the comparative methods. The findings demonstrate that PCPrior consistently outperforms all the test prioritization methods across various parameter configurations, as evident from the visual analysis of Figure 6.4. Furthermore, it can be observed 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 PCPrior. PCPrior exhibits relative stability when this parameter is adjusted. Conversely, the parameters *max_depth* (representing the maximum tree depth for each LightGBM model) and *learning_rate* (referring to the boosting learning rate) have a relatively larger influence on the effectiveness of PCPrior. Remarkably, regardless of the extent to which the parameters influence PCPrior's effectiveness, we see that PCPrior can consistently outperform all the compared methods across different parameter settings.

Answer to RQ3: *PCPrior consistently outperforms other test prioritization methods across various parameter settings. The parameter colsample_bytree has a minor impact on PCPrior's effectiveness, while the parameters max_depth and learning_rate have a relatively larger impact. However, despite these fluctuations, PCPrior consistently remains more effective than the comparative methods.*

6.5.4 RQ4: Effectiveness on Noisy Test Inputs

Objectives: We further investigate the effectiveness of PCPrior and its variants on noisy data.

Experimental design: In the initial phase, we introduce noise to the original 3D point cloud datasets, namely ModelNet40, ShapeNet, and S3DIS, to create noisy data. To generate a noisy dataset from an initial test set denoted as T, each test instance



a) ModelNet, max_depth **b)** ModelNet, cols_bytree **c)** ModelNet, learning_rate

d) S3DIS, max_depth **e)** S3DIS, cols_bytree **f)** S3DIS, learning_rate

g) ShapeNet, max_depth **h)** ShapeNet, cols_bytree **i)** ShapeNet, learning_rate

Figure 6.4: Impact of main parameters in PCPrior

$t \in T$ undergoes a specific modification. Specifically, within each test instance t (a point cloud), approximately 30% of the points are subjected to a random offset in the x, y, and z coordinates, while the remaining 70% of the points remain unaltered. The decision to select 30% of the points in a point cloud for displacement is because: if a large number of the points were to be shifted, it would lead to a significant number of tests being misclassified by the original model. In such a scenario, all test prioritization methods could identify a large number of misclassified tests. This, in turn, could affect the evaluation of PCPrior. Therefore, we opted to carefully select the modification ratio that is not excessively high for the evaluation of PCPrior. As a result, we generate ten noisy datasets for each original dataset, resulting in a total of 30 (3×10) noisy datasets. Each of these noisy datasets is paired with five different models, resulting in a total of 150 (30×5) subjects. Finally, we compared the effectiveness of PCPrior, its variants, and all the comparative test prioritization approaches on the generated 150 noisy subjects. On the generated noise subjects, we assessed the effectiveness of PCPrior, the confidence-based test prioritization methods, along with PCPrior variants that employed Logistic Regression [251], XGBoost [13], Random Forest [90], DNNs [257], and TabNet [258] as ranking models, respectively. We also included random selection as a baseline for comparison.

Statistical analysis Similar to RQ1, due to the inherent randomness in the model training process, we performed the experiments ten times and conducted a statistical analysis. Like in RQ1, the statistical analysis method we used is the paired two-sample t-test [104]. We calculated the p-value and effect size for the experimental results. We consider that if the p-value is less than 10^{-05}, the difference between the two sets of data is statistically significant [105]. Moreover, to ensure that the difference

Table 6.10: Effectiveness comparison among PCPrior and the compared approaches in terms of the average APFD values on noisy datasets

Approach	ModelNet					S3DIS					ShapeNet				
	DGCNN	PointConv	MSG	SSG	PointNet	DGCNN	PointConv	MSG	SSG	PointNet	DGCNN	PointConv	MSG	SSG	PointNet
Random	0.501	0.501	0.501	0.499	0.502	0.501	0.503	0.500	0.501	0.499	0.499	0.499	0.499	0.499	0.499
DeepGini	0.743	0.695	0.700	0.679	0.708	0.587	0.542	0.555	0.533	0.592	0.752	0.641	0.677	0.718	0.642
VanillaSM	0.750	0.698	0.705	0.686	0.712	0.588	0.542	0.555	0.533	0.594	0.758	0.644	0.685	0.723	0.647
PCS	0.754	0.698	0.707	0.688	0.714	0.585	0.541	0.551	0.531	0.594	0.762	0.643	0.693	0.728	0.650
Entropy	0.728	0.685	0.688	0.666	0.697	0.582	0.540	0.553	0.532	0.587	0.735	0.633	0.661	0.704	0.633
PCPrior^{L}	0.782	0.745	0.732	0.717	0.728	0.610	0.568	0.597	0.617	0.636	0.785	0.737	0.794	0.824	0.670
PCPrior^{X}	0.793	0.761	0.767	0.754	0.765	0.726	0.667	0.687	0.663	0.751	0.864	0.774	0.838	0.856	0.787
PCPrior^{R}	0.779	0.742	0.741	0.725	0.739	0.693	0.655	0.676	0.654	0.725	0.825	0.750	0.822	0.838	0.764
PCPrior^{D}	0.782	0.748	0.747	0.727	0.741	0.647	0.633	0.651	0.639	0.672	0.838	0.765	0.821	0.839	0.773
PCPrior^{T}	0.766	0.739	0.746	0.730	0.743	0.654	0.637	0.659	0.641	0.694	0.856	0.772	0.830	0.848	0.785
PCPrior	0.794	0.762	0.770	0.755	0.766	0.728	0.668	0.690	0.665	0.753	0.862	0.776	0.837	0.855	0.788

between the results of PCPrior and the compared approach is non-negligible, the effect size should be greater than or equal to 0.2.

Results: The experimental results for RQ4 are presented in Table 6.10, Table 6.11, Table 6.12, Table 6.13, Table 6.14, and Figure 6.5. Specifically, Table 6.10 and Table 6.11 provide a comparative analysis of the effectiveness of PCPrior (including its variants) and various test prioritization methods in the context of noisy data, using the APFD metric. On the other hand, Table 6.13 and Table 6.14 present the comparative evaluation based on the PFD metric.

Table 6.11: Performance improvement of PCPrior over the compared approaches in terms of APFD on 150 noisy subjects

Approach	# Best cases	Average APFD	Improvement(%)
Random	0	0.500	53.00
DeepGini	0	0.651	17.51
VanillaSM	0	0.655	16.79
PCS	0	0.656	16.62
Entropy	0	0.642	19.16
PCPrior^{L}	0	0.703	-
PCPrior^{X}	35	0.763	-
PCPrior^{R}	0	0.742	-
PCPrior^{D}	0	0.735	-
PCPrior^{T}	0	0.740	-
PCPrior	115	0.765	-

Table 6.10 shows the comparison results of PCPrior, its variants, and comparative methods on noisy test inputs in terms of APFD. We found that the effectiveness of PCPrior and its variants surpasses that of all compared test prioritization methods in each case. Specifically, the APFD values for PCPrior range from 0.665 to 0.862. For PCPrior's variants, the APFD values range from 0.568 to 0.864. For the compared test prioritization methods, the APFD values range from 0.499 to 0.762. Furthermore, Table 6.11 provides a more detailed analysis by presenting the number of cases in which each test prioritization method performs the best, the average APFD value, and the improvement of PCPrior relative to each comparative method. We see that, on noisy test inputs, PCPrior's average APFD is 0.765, while the range for its variants is 0.703 to 0.763. The average APFD range for the benchmark methods is 0.500 to 0.656. Notably, PCPrior performs the best in 76.7% (115 out of 150)

Table 6.12: Statistical analysis on noisy test inputs (in terms of p-value and effect size)

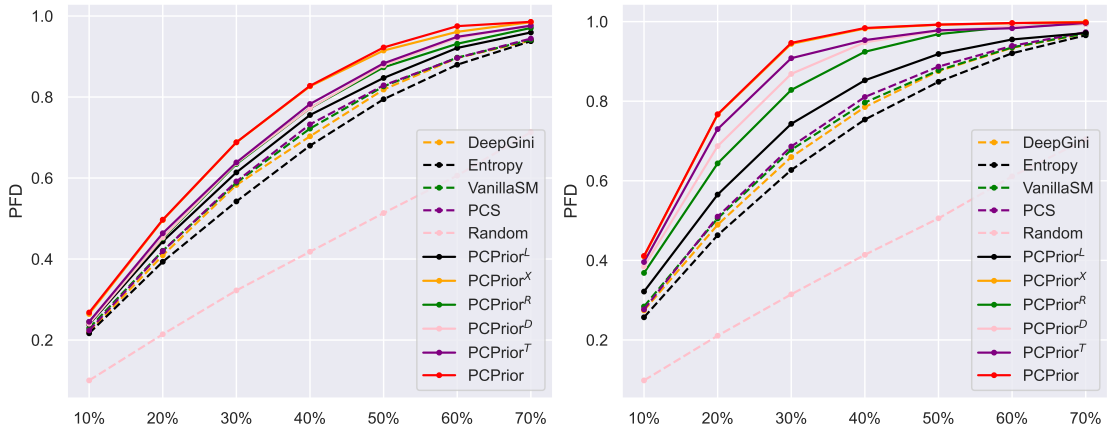
	Random	DeepGini	VanillaSM	PCS	Entropy
PCPrior (p-value)	1.156×10^{-10}	1.688×10^{-08}	3.521×10^{-08}	5.049×10^{-08}	3.385×10^{-09}
PCPrior (effect size)	4.329	2.958	2.792	2.713	3.352

Table 6.13: Effectiveness comparison of PCPrior and the compared approaches in terms of the PFD values on noisy datasets

Data	Approach	#Best cases in PFD				Average PFD			
		PFD-10	PFD-20	PFD-30	PFD-40	PFD-10	PFD-20	PFD-30	PFD-40
ModelNet	Random	0	0	0	0	0.099	0.201	0.300	0.402
	DeepGini	0	0	0	0	0.219	0.404	0.564	0.701
	VanillaSM	0	0	0	0	0.225	0.417	0.577	0.712
	PCS	0	0	0	0	0.220	0.416	0.583	0.719
	Entropy	0	0	0	0	0.210	0.387	0.542	0.677
	PCPrior^{L}	1	0	0	0	0.246	0.459	0.635	0.772
	PCPrior^{X}	19	17	20	13	0.264	0.494	0.687	0.830
	PCPrior^{R}	1	1	0	0	0.252	0.465	0.641	0.777
	PCPrior^{D}	0	0	0	0	0.250	0.468	0.647	0.787
	PCPrior^{T}	0	0	0	0	0.252	0.470	0.651	0.790
	PCPrior	29	32	30	37	0.265	0.497	0.689	0.833
S3DIS	Random	0	0	0	0	0.099	0.201	0.302	0.402
	DeepGini	0	0	0	0	0.133	0.258	0.375	0.486
	VanillaSM	0	0	0	0	0.135	0.260	0.377	0.489
	PCS	0	0	0	0	0.130	0.255	0.373	0.485
	Entropy	0	0	0	0	0.131	0.254	0.370	0.480
	PCPrior^{L}	0	0	0	0	0.151	0.291	0.419	0.541
	PCPrior^{X}	8	8	6	7	0.188	0.369	0.536	0.689
	PCPrior^{R}	0	0	0	0	0.182	0.350	0.509	0.654
	PCPrior^{D}	0	0	0	0	0.173	0.331	0.476	0.608
	PCPrior^{T}	0	0	0	0	0.176	0.339	0.488	0.623
	PCPrior	42	42	44	43	0.189	0.370	0.539	0.692
ShapeNet	Random	0	0	0	0	0.099	0.199	0.298	0.399
	DeepGini	0	0	0	0	0.212	0.390	0.544	0.675
	VanillaSM	0	0	0	0	0.221	0.403	0.557	0.685
	PCS	0	0	0	0	0.221	0.411	0.568	0.694
	Entropy	0	0	0	0	0.201	0.372	0.519	0.649
	PCPrior^{L}	0	0	0	0	0.277	0.519	0.703	0.822
	PCPrior^{X}	17	24	39	30	0.317	0.616	0.841	0.951
	PCPrior^{R}	0	0	0	0	0.303	0.567	0.769	0.894
	PCPrior^{D}	0	0	0	0	0.308	0.585	0.792	0.912
	PCPrior^{T}	0	0	0	0	0.314	0.606	0.826	0.939
	PCPrior	33	26	11	20	0.318	0.614	0.838	0.949

Table 6.14: Average effectiveness comparison of PCPrior and the compared approaches in terms of the PFD values on noisy datasets

Approach	#Best cases in PFD				Average PFD			
	PFD-10	PFD-20	PFD-30	PFD-40	PFD-10	PFD-20	PFD-30	PFD-40
Random	0	0	0	0	0.099	0.201	0.300	0.401
DeepGini	0	0	0	0	0.188	0.351	0.494	0.621
VanillaSM	0	0	0	0	0.194	0.36	0.504	0.629
PCS	0	0	0	0	0.190	0.361	0.508	0.633
Entropy	0	0	0	0	0.181	0.337	0.477	0.602
PCPrior^{L}	1	0	0	0	0.225	0.423	0.585	0.712
PCPrior^{X}	44	49	65	50	0.256	0.493	0.688	0.823
PCPrior^{R}	1	1	0	0	0.246	0.461	0.639	0.775
PCPrior^{D}	0	0	0	0	0.244	0.461	0.639	0.769
PCPrior^{T}	0	0	0	0	0.248	0.472	0.655	0.784
PCPrior	104	100	85	100	0.257	0.494	0.689	0.825



a) ModelNet (noisy), PointNet **b)** ShapeNet (noisy), DGCNN

Figure 6.5: Test prioritization effectiveness among PCPrior and the compared approaches for ModelNet(Noisy) with PointNet and ShapeNet(Noisy) with DGCNN on noisy datasets. X-Axis: the percentage of prioritized tests; Y-Axis: the percentage of detected misclassified tests.

of the cases, while PCPrior^X performs the best in 23.3% (35 out of 150) of the cases. PCPrior continues to outperform the variants of PCPrior that utilize DNN ranking models (PCPrior^T and PCPrior^N) in all cases. Moreover, PCPrior shows an improvement ranging from 16.62% to 53.00% over all the comparative methods. The above experimental results demonstrate that, under the APFD measurement, the average effectiveness of PCPrior surpasses all its variants and comparative methods on noisy datasets.

The results from the statistical analysis on noisy test inputs are presented in Table 6.12. We see that the p-values for the experimental results of PCPrior and each of the compared methods are all less than 10^{-05}, indicating that PCPrior statistically outperforms all the test prioritization methods on noisy datasets. For instance, the p-value between PCPrior and DeepGini is 1.688×10^{-08}. The p-value between PCPrior and PCS is 5.049×10^{-08}. Furthermore, all the effect sizes of PCPrior and the compared approaches exceed 0.2, demonstrating a non-negligible difference. Notably, all the effect sizes are greater than 0.8. For example, the effect size between PCPrior and VanillaSM is 2.792, and the effect size between PCPrior and Entropy is 3.352. According to Cohen's d [106], this implies that the difference in experimental results between PCPrior and the compared methods is not only statistically significant but also relatively "large" in scale.

Table 6.13 and Table 6.14 present a comparative analysis regarding the PFD metric. It is observed that in Table 6.13, the best performance is consistently achieved by PCPrior or its variants across all cases. Table 6.14 provides a deeper analysis of this finding. When considering different percentages of test data prioritization, PCPrior consistently outperforms other approaches in terms of effectiveness, as evidenced by the highest number of best-performing cases and the highest average PFD values. Figure 6.5 visually demonstrates the performance comparison of PCPrior, its variants, and the comparative methods on noisy data. The solid lines depict PCPrior and its variant methods, while the dashed lines represent the comparative methods. We see that across the noisy dataset, PCPrior and all its variants exhibit higher effectiveness compared to all comparative methods. Furthermore, PCPrior demonstrates superior performance when compared to its variants.

Answer to RQ4: *PCPrior consistently exhibits superior performance in comparison to all the test prioritization approaches considered in the context of noisy data, as evaluated by APFD and PFD. Notably, the average improvement achieved in terms of APFD ranges from 16.62% to 53.00%, highlighting the significant effectiveness of PCPrior over the compared methods. Furthermore, PCPrior consistently outperforms its variants in a majority of cases.*

6.5.5 RQ5: Feature contribution analysis

Objectives: We investigate the contributions of each type of features on the effectiveness of PCPrior for test prioritization. Our investigation revolves around two primary sub-questions, as outlined below:

- **RQ-5.1** Based on the ablation study, to what extent does each type of features contribute to the effectiveness of PCPrior?
- **RQ-5.2** What is the distribution of feature types among the top-N most contributing features towards PCPrior?

Experimental design: We conduct two experiments below to answer the above two sub-questions.

[**Experiment ❶**] In the original PCPrior framework, a comprehensive set of four feature types is generated, namely mutation features (MF), spatial features (SF), uncertainty features (UF), and prediction features (PF). To compare the contributions of each feature type on PCPrior's effectiveness, we conducted a carefully designed ablation study following the prior work [107]. More specifically, we individually removed one type of features and evaluated PCPrior's effectiveness under these modified conditions. For instance, to assess the contribution of SF features, PCPrior is executed with SF features excluded while retaining the other three feature types. The resulting performance of PCPrior is then evaluated under these adjusted circumstances. Similarly, to gauge the contribution of MF features, PCPrior is executed without generating MF features while keeping generating the other three feature types. The performance of PCPrior is subsequently assessed in this context. By conducting the aforementioned ablation study, we can determine the contribution of each feature type to the overall effectiveness of PCPrior.

[**Experiment ❷**] The method we employed to evaluate the contributions of features is the cover metric within the XGBoost algorithm [13]. Initially, we utilized the cover metric to compute the importance scores of each feature used by PCPrior for test prioritization. Subsequently, we selected the top-N most important features based on these scores. By analyzing the categorization of these features, we investigated the contributions of different feature types to the effectiveness of PCPrior. Below, we provide an overview of how XGBoost quantifies feature importance.

The cover metric employed in XGBoost serves as a means to quantify the importance of features by assessing the average coverage of individual instances across the leaf nodes within a decision tree. This metric operates by evaluating the frequency with which a specific feature is utilized for partitioning the data across the entirety of the ensemble's trees. The coverage values associated with each feature across all trees are subsequently aggregated, resulting in a cumulative coverage value. To obtain the average coverage of each instance by the leaf nodes, the cumulative coverage value is normalized in relation to the total number of instances. Consequently, the derived coverage value of a given feature plays a crucial role in determining its significance, with features that demonstrate higher coverage values

Table 6.15: Ablation study on different features of PCPrior: Mutation Features(MF), Spatial Features(SF), Uncertainty Features(UF), Prediction Features(PF). 'w/o' means 'without'

Approach	Dataset			Average
	ModelNet	S3DIS	ShapeNet	
PCPrior w/o MF	0.788	0.811	0.875	0.825
PCPrior w/o SF	0.769	0.699	0.841	0.769
PCPrior w/o UF	0.785	0.816	0.871	0.824
PCPrior w/o PF	0.782	0.778	0.874	0.811
PCPrior	0.797	0.822	0.890	0.836

being considered more important.

Results: The experimental results of RQ5.1 are presented in Table 6.15. In this table, 'w/o' stands for 'without.' For example, 'PCPrior w/o SF' refers to executing PCPrior without generating the spatial features. From Table 6.15, we see that the original PCPrior achieves the highest average effectiveness. Removing any type of feature results in a decrease in the effectiveness of PCPrior, demonstrating that each type of features contributes to PCPrior's effectiveness. For instance, on the Modelnet dataset, the average APFD value of the original PCPrior is 0.797. Removing spatial features results in a decline of PCPrior's average APFD to 0.769, while the removal of mutation features causes a decrease to 0.788, uncertainty features to 0.785, and prediction features to 0.782.

Furthermore, among all four types of features, spatial features demonstrate the highest average contributions. This inference is drawn from the following findings: When removing spatial features, PCPrior's effectiveness shows the largest average decrease. Specifically, when removing spatial features (SF features), the average APFD decreases by 0.067. In comparison, the removal of mutation features (MF) leads to an average APFD decrease of 0.011, uncertainty features (UF) result in an average APFD decrease of 0.012, and prediction features (PF) show an average APFD decrease of 0.025. Moreover, across all datasets, removing spatial features results in the highest average decrease in PCPrior's effectiveness.

Answer to RQ5.1: *The ablation study demonstrates that each type of features contributes to the effectiveness of PCPrior. Moreover, spatial features show the highest average contributions.*

The findings of RQ5.2 are presented in Table 6.16, where the scores represent the importance levels of each feature. For each combination of model and dataset, we present the top-N features that contribute the most. It is worth noting that abbreviations SF, MF, PF, and UF are used to represent spatial features, mutation features, prediction features, and uncertainty features, respectively. Moreover, the numbers after the feature abbreviations indicate the indices of the corresponding features. For instance, *SF-23* represents the spatial feature with index 23. From Table 6.16, it can be observed that all four types of features consistently appear among the top-N most contributing features across various subjects. As an example, in the case of the PointConv subject with the S3DIS dataset, SF features account for 50%, UF features account for 30%, MF features account for 10%, and PF features account for 10%. Remarkably, among the 15 subjects investigated, in 93.3% (14 out of 15) of the cases, the top 10 contributing features include three or more distinct

Table 6.16: Top-10 most contributing features on the effectiveness of PCPrior

Data	Rank	DGCNN		PointConv		MSG		SSG		PointNet	
		Feature	Value	Feature	Value	Feature	Value	Feature	Value	Feature	Value
ModelNet	1	SF-23	348	MF-132	272	SF-f23	271	SF-46	261	MF-120	309
	2	MF-141	203	MF-126	261	PF-112	260	MF-147	238	PF-112	276
	3	MF-143	197	SF-52	243	MF-120	210	MF-114	225	MF-144	265
	4	PF-112	195	PF-112	221	MF-143	173	PF-112	224	SF-52	256
	5	MF-137	187	MF-125	219	MF-127	167	MF-131	199	SF-28	210
	6	MF-145	178	MF-139	214	SF-52	163	MF-127	196	PF-90	174
	7	SF-22	171	MF-129	213	SF-28	160	MF-122	155	SF-69	167
	8	MF-123	169	MF-135	207	MF-135	141	SF-42	142	SF-1	166
	9	MF-131	167	UF-118	206	SF-f22	138	SF-29	131	MF-139	165
	10	MF-136	155	MF-124	201	PF-88	134	SF-25	128	SF-25	164
S3DIS	1	UF-85	166	SF-18	168	UF-85	175	UF-87	190	UF-87	156
	2	PF-75	112	UF-85	140	UF-87	151	SF-55	161	UF-85	119
	3	SF-61	111	SF-61	127	UF-86	115	UF-86	148	SF-65	115
	4	SF-60	106	UF-87	123	SF-28	114	SF-19	128	SF-68	106
	5	SF-62	105	MF-106	100	MF-91	111	SF-20	110	PF-74	105
	6	SF-2	94	PF-80	99	PF-73	110	UF-90	107	PF-82	94
	7	UF-86	93	SF-60	99	SF-62	107	PF-84	102	SF-17	87
	8	PF-74	86	SF-62	93	SF-16	102	SF-65	101	UF-90	86
	9	SF-35	83	SF-4	92	MF-95	98	MF-99	99	PF-79	86
	10	SF-16	82	UF-86	90	PF-74	96	PF-78	98	PF-80	85
ShapeNet	1	PF-122	908	MF-135	1223	UF-122	952	UF-122	984	UF-122	1021
	2	MF-151	554	MF-141	1059	UF-124	518	MF-156	536	PF-101	502
	3	MF-148	542	UF-130	934	MF-155	505	UF-124	527	PF-100	435
	4	MF-140	539	MF-153	850	MF-145	503	SF-70	496	PF-110	523
	5	MF-145	498	MF-143	801	MF-136	475	PF-107	477	PF-94	415
	6	SF-42	486	PF-122	798	PF-121	467	SF-42	476	SF-71	398
	7	PF-107	486	MF-154	776	MF-153	460	MF-149	435	UF-124	391
	8	PF-116	424	MF-150	749	UF-126	446	MF-155	408	SF-42	389
	9	PF-109	423	MF-148	725	SF-42	446	PF-104	405	PF-87	383
	10	PF-110	376	PF-121	701	SF-71	429	MF-131	404	PF-90	371

feature types. These experimental findings provide robust evidence that all three feature categories play pivotal roles in the effectiveness of PCPrior.

Answer to RQ5.2: *All four types of features, namely spatial features, mutation features, uncertainty features, and prediction features, exhibit consistent presence among the top-N most influential features across diverse subjects.*

6.5.6 RQ6: Retraining 3D shape classification models with PCPrior and uncertainty-based methods

Objectives: We investigate whether PCPrior and uncertainty-based test prioritization approaches are effective in selecting informative retraining inputs to enhance the performance of a 3D shape classification model.

Experimental design: Building on the previous research [105], we structured our retraining experiments in the following manner. First, we randomly divided the point cloud dataset into three parts: the training set, the candidate set, and the test set, in a 4:4:2 ratio. The candidate set was used for retraining, while the test set was reserved for evaluation purposes and remained untouched. In the first phase, we trained a 3D shape classification model using only the initial training set. In the second round, we integrate an extra 10% of new inputs from the candidate set into the current training set without replacement. The chosen inputs for inclusion are those prioritized in the top 10% by PCPrior and the compared test prioritization approaches. The prioritization range we selected for retraining is from 10% to 70%. We chose this range because, according to the experimental results (cf. Section 6.5.1), when prioritizing up to 70%, PCPrior can identify the majority of misclassified inputs in the dataset (99.6%), as indicated in Table 6.6. For example, in the ShapeNet dataset, within the 70% prioritized test set, PCPrior has identified 99.8% of misclassified inputs. Given that the primary objective of this research question is to validate PCPrior's effectiveness in retraining, we chose a retraining range of up to 70%. Following prior work [105], we retrained the model using the expanded training set, ensuring equal treatment of both old and new training data. This retraining was repeated in five rounds. The reason for opting to conduct retraining five times is that the training process of DNN models involves various random factors, and conducting multiple rounds of retraining can contribute to ensuring the stability and reproducibility of the results. On the other hand, excessive retraining can lead the model to over-optimize for a specific dataset, resulting in overfitting. Therefore, based on the experimental experience of existing studies [259], we choose to conduct five rounds of retraining. To account for the inherent randomness in model training, we repeated all experiments three times and reported the average results across these repetitions.

Results: The experimental results for RQ6 are presented in Table 6.17, which illustrates the average accuracy of 3D shape classification models after retraining. In each case, we have highlighted the approach with the highest effectiveness in grey for a quick and straightforward interpretation of the findings. As shown in Table 6.17, PCPrior and all uncertainty-based approaches demonstrate better average effectiveness compared to random selection. However, the improvements they achieved are relatively small. For instance, when selecting 10% of tests for retraining the original model, PCPrior's selected samples result in a post-retrain model accuracy of 0.851, while uncertainty-based methods range from 0.846 to 0.850. In contrast, the

Table 6.17: The average accuracy value after retraining with 10%~70% prioritized tests

Approach	Accuracy of percentage of datasets							Average
	10%	20%	30%	40%	50%	60%	70%	
Random	0.847	0.855	0.865	0.872	0.879	0.886	0.887	0.870
DeepGini	0.846	0.866	0.872	0.880	0.889	0.896	0.898	0.878
VanillaSM	0.850	0.867	0.870	0.881	0.890	0.891	0.898	0.878
PCS	0.846	0.861	0.868	0.884	0.888	0.893	0.898	0.877
Entropy	0.846	0.861	0.869	0.881	0.886	0.895	0.896	0.876
PCPrior	0.851	0.868	0.873	0.883	0.888	0.898	0.901	0.880

random selection yields an accuracy of 0.847. Similarly, when choosing 70% of tests for retraining the original model, PCPrior's selected samples result in a post-retrain model accuracy of 0.901, while uncertainty-based methods range from 0.896 to 0.898. In comparison, the random selection yields an accuracy of 0.887.

The reasons for the aforementioned findings, where PCPrior and uncertainty-based methods show only small improvements over random selection in enhancing model accuracy, include:

- **Lack of Diversity**: PCPrior and uncertainty-based methods focus on identifying corner cases, which are tests that the model finds more challenging. Consequently, the tests identified can lack diversity. In contrast, random selection provides a broader and more diverse set of samples, contributing to the model learning more comprehensive data features and thereby improving its generalization capability.
- **Overfitting Risk**: Concentrating on samples the model is most likely to predict incorrectly can lead to overfitting. These samples can exhibit certain extreme or uncommon features, causing the model to overly adapt to these specific cases after retraining and ignoring more widespread patterns.

Moreover, another observation from the results in Table 6.17 is that PCPrior performs better than uncertainty-based methods on average. Specifically, PCPrior performs the best in 75% (6 out of 8) cases, while uncertainty-based methods perform the best in only 25% (2 out of 8) cases. Moreover, after retraining the original model with tests selected by PCPrior, the average accuracy of the resulting model is 0.880. In contrast, for uncertainty-based methods, the range is from 0.876 to 0.878.

Answer to RQ6: *PCPrior and uncertainty-based methods perform better than the random selection approach. However, the improvement achieved is relatively modest, suggesting that these prioritization approaches, aimed at identifying potentially misclassified tests, can guide the retraining of 3D shape classification models but with limited effectiveness. Additionally, PCPrior demonstrates better effectiveness compared to uncertainty-based test prioritization methods.*

6.6 Discussion

6.6.1 Limitations of PCPrior

PCPrior suffers from a notable limitation regarding its ability to ensure the diversity of the selected data, which has also been recognized in previous investigations on uncertainty-based test prioritization techniques [6]. This concern arises from the fact that neither PCPrior nor these earlier approaches account for diversity

during the process of prioritizing test inputs. However, despite this shared limitation, PCPrior has demonstrated considerable effectiveness in identifying a substantial majority of misclassified test inputs by leveraging a small proportion of prioritized test cases. The experimental results illustrate that PCPrior can detect over 95% of misclassified tests on natural datasets by prioritizing a mere 50% of the test inputs. This noteworthy performance highlights PCPrior's ability to efficiently identify a significant proportion of misclassified tests using a reduced set of prioritized tests, even without explicitly ensuring the diversity. While prioritizing diverse misclassified tests undoubtedly enhances overall testing quality, in practical scenarios with limited time and resource constraints, prioritizing a significant majority of misclassified tests can still be a viable strategy. Therefore, the capacity of PCPrior to identify a significant proportion of misclassified tests while operating within the constraints of a reduced number of prioritized tests becomes particularly advantageous in situations where time and resources are scarce.

Another limitation is that PCPrior is specifically designed for classification models and cannot be adapted for regression models. This is primarily due to two reasons: 1) PCPrior requires generating mutation features from tests for test prioritization. However, for a given test, generating mutation features involves comparing whether the model's predictions for this test and its variants are the same. This approach is not applicable to regression models because the predictions of regression models are continuous numerical values. 2) PCPrior requires generating prediction features and uncertainty features for test prioritization. For a given test, the generation of these two types of features requires the model to predict the probabilities of this test belonging to each category. Therefore, PCPrior cannot be applied to regression models.

6.6.2 Generality of PCPrior

Our experimental findings have validated the effectiveness of PCPrior based on a large number of subjects, encompassing both natural and noisy scenarios. Although our study initially focused on three datasets, PCPrior can be generalized to a broader range of 3D shape classification domains. The adaptability of PCPrior stems from its core process, which is the generation of four types of features: spatial features, mutation features, prediction features, and uncertainty features. PCPrior can perform test prioritization through an automated pipeline when the evaluated model and dataset meet the criteria for generating these four types of features. Below, we provide a detailed explanation of the specific conditions that the evaluated model and dataset require to meet in order to utilize PCPrior:

- **Requirement 1: Point Cloud Dataset.** The generation of spatial features and mutation features requires the dataset to be a point cloud dataset. This is because these two features are specifically tailored for point cloud data. For a given point cloud dataset, PCPrior can automatically generate its spatial feature and mutation features.
- **Requirement 2: Classification Tasks.** The generation of the prediction features and uncertainty features necessitates that both the model and the dataset be oriented toward classification tasks. This is because these two types of features are generated from the model's predictions for each test within the test set. Specifically, for a given test, the generation of these two types of features requires the model to predict the probabilities of this test belonging to each category.

Models and datasets that meet the above conditions can use PCPrior for test prioritization, making PCPrior widely applicable in a diverse range of 3D shape classification tasks.

6.6.3 Threats to Validity

6.6.3.1 Internal Threats to Validity.

Internal threats to validity primarily arise from the implementation of our proposed PCPrior methodology and the compared approaches. To address these threats, we implemented PCPrior using the widely adopted PyTorch library. Additionally, we utilized the original implementations of the compared approaches as provided by their respective authors, minimizing potential implementation biases. Another internal threat emerges from the inherent randomness associated with model training. To mitigate this threat and ensure the stability of our experimental results, we conducted a statistical analysis. Specifically, we performed ten repetitions of the training process and calculated the statistical significance of the experimental results, thereby reducing the influence of randomness.

6.6.3.2 External Threats to Validity.

External threats to validity primarily reside in the 3D point cloud dataset and DNN models employed in our study. To mitigate these threats, we adopted a large number of subjects, encompassing both natural and noisy data, thus ensuring a comprehensive exploration of various scenarios. By including diverse data types, we aimed to enhance the robustness and generalizability of our findings. As a future direction, we aim to extend the application of PCPrior to 3D point cloud datasets characterized by diverse properties, thereby broadening the scope and applicability of our proposed methodology.

6.7 Related Work

6.7.1 Test Prioritization Techniques

Test prioritization aims to determine the optimal order for executing test cases, thereby enabling the early detection of system bugs. The idea was first mentioned by Wong et al. [260]. In field of conventional software engineering [77, 166, 167, 261, 79], several corresponding studies have been conducted. Di Nardo *et al.* [79] conducted a study evaluating the effectiveness of coverage-based prioritization strategies using real-world regression faults. Their research shed light on the efficiency of different techniques in detecting bugs. Henard *et al.* [77] conducted a comprehensive investigation to compare existing test prioritization approaches, specifically focusing on white-box and black-box strategies. Their findings revealed minimal distinctions between these two categories of strategies. Chen *et al.* [167] proposed the LET (Learning-based and Execution Time-aware Test prioritization) technique for prioritizing test programs in compiler testing, demonstrating its effectiveness. LET employs a learning process to identify program features and predict the bug-revealing probability of new test programs, along with a scheduling process that prioritizes test programs based on their bug-revealing probabilities.

Furthermore, several studies have focused on addressing the test prioritization

problem using mutation testing techniques [56, 57, 58, 111, 47]. Shin *et al.*[57] proposed a diversity-aware mutation adequacy criterion to guide test case prioritization and empirically evaluated mutation-based prioritization techniques using large-scale developer-written test cases. Papadakis *et al.*[58] introduced the concept of mutating Combinatorial Interaction Testing models and prioritizing tests based on their ability to detect mutants. They demonstrated a strong correlation between the number of model-based mutants killed and code-level faults detected by the test cases.

Regarding test prioritization for DNNs, Feng *et al.*[6] proposed DeepGini, which identifies possibly misclassified tests based on model uncertainty. DeepGini assumes that a test is more likely to be mispredicted if the DNN outputs similar probabilities for each class. Weiss *et al.*[9] conducted a comprehensive investigation of various DNN test input prioritization techniques, including several uncertainty-based metrics such as Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy. Moreover, Wang *et al.* [10] developed PRIMA, an intelligent mutation analysis-based approach, specifically tailored for prioritizing test inputs in DNNs. However, the mutation rules of PRIMA are not adapted to handle 3D point data, which constitutes unstructured sets of points in three-dimensional space. To address this limitation, we propose PCPrior, a novel test prioritization technique that is specifically designed for 3D point cloud data. PCPrior effectively generates a set of features to facilitate test prioritization.

6.7.2 Mutation Testing for DNNs

In the field of mutation testing for DNNs, various studies [61, 20, 59, 111, 60] have been conducted, focusing on the development of different mutation operators and frameworks. Shen *et al.* introduced MuNN [61], a mutation analysis method specifically designed for neural networks. MuNN defined five mutation operators based on the characteristics of neural networks. The research findings highlighted that mutation analysis exhibited strong domain-specific characteristics, indicating the necessity of domain-specific mutation operators to enhance the analysis process. Ma *et al.* [20] proposed DeepMutation, a methodology for assessing the quality of test data in DL systems using mutation testing. They devised a collection of source-level and model-level mutation operators to introduce faults into the training data, training programs, and DL models. Subsequently, Hu *et al.* [59] extended DeepMutation to DeepMutation++ by introducing a new set of mutation operators for feed-forward neural networks (FNNs) and Recurrent Neural Networks (RNNs) and enabled dynamic mutation of run-time states in RNNs. Jahangirova *et al.* [111] conducted a comprehensive empirical study on the DL mutation operators in the existing literature. Their investigation shed light on the necessity for a stochastic definition of mutation killing. Furthermore, they successfully identified a subset of mutation operators that exhibit high effectiveness, along with the associated configurations that yield the highest efficacy. Humbatova *et al.* presented DeepCrime [60], the first mutation testing tool that implemented a set of DL mutation operators based on real DL faults. This tool provided a comprehensive framework for evaluating the robustness and fault tolerance of DNNs.

6.7.3 Deep Neural Network Testing

In addition to test input prioritization, test selection [105] is another approach for improving the efficiency of DNN testing. The goal of test selection is to estimate

the accuracy of the entire set by only labeling a selected subset of test inputs, thereby reducing the labeling cost associated with DNN testing. Several effective test selection methods have been proposed in the literature [46, 48, 49, 105, 262]. Li *et al.*[48] introduced Cross Entropy-based Sampling (CES), a method for selecting a representative subset of test inputs to estimate the accuracy of the entire testing set. CES minimizes the cross-entropy between the selected set and the original test set to ensure that the distribution of the selected test set is similar to that of the original set. Chen *et al.*[46] proposed Practical Accuracy Estimation (PACE) for test selection. The basic principle of PACE involves clustering all the tests in the test set and using the MMD-critic algorithm [49] to perform prototype selection. For the remaining test inputs that do not belong to any group, adaptive random testing is employed for test selection.

In addition to focusing on improving the efficiency of DNN testing, many studies in the field of DNN testing [8, 7, 20, 52, 59, 51] concentrate on measuring the adequacy of DNNs. Pei *et al.*[8] proposed neuron coverage, a metric for evaluating how well a test set covers the logic of a DNN model. Ma *et al.*[7] introduced DeepGauge, a set of coverage criteria to measure the test adequacy of DNNs. DeepGauge considers neuron coverage as an important indicator of the effectiveness of a test input. Moreover, they proposed new metrics with different granularities based on neuron coverage to differentiate adversarial attacks from legitimate test data. Kim *et al.* [52] proposed surprise adequacy as a measure of identifying the effectiveness of a test input within a test set. Surprise adequacy focuses on measuring the surprise of a test input with respect to the training set, where surprise is defined as the difference in the activation value of neurons when faced with this new test input. Dola *et al.* [53] proposed the Input Distribution Coverage (IDC) framework to evaluate the black-box test adequacy of DNNs. The framework utilizes a Variational Autoencoder (VAE) to transform test inputs into feature vectors, establishing a coverage domain. Within this domain, Combinatorial Interaction Testing (CIT) metrics are applied to measure test coverage. Riccio *et al.* [263] introduced the notion of "mutation adequacy" to assess the effectiveness of test sets in identifying artificially injected faults (mutations) in deep learning systems. Moreover, they proposed DEEPMETIS as a solution to enhance the mutation adequacy of the test set (i.e., improving the test set's ability to detect mutations).

Furthermore, several studies focused on utilizing the decision boundary to enhance the quality assurance of DL-based software. Riccio *et al.* [264] proposed the notion of the "frontier of behaviors" referring to the inputs at which a DL system begins to exhibit misbehavior. This concept serves as a metric for evaluating the quality of DL systems. The assessment involves determining whether the frontier of misbehaviors extends beyond the system's validity domain, in which case the quality check is deemed successful. Conversely, if the frontier intersects with the validity domain, it indicates quality deficiencies in the system. Biagiola *et al.* [265] introduced an innovative approach to assessing the adaptability of reinforcement learning (RL) systems, focusing on their capacity to adjust to dynamic environments. Their method involves computing the adaptation boundary within a changing environment and presenting them through two-dimensional or multi-dimensional adaptability/anti-regression heatmaps. These visualizations serve to quantify the system's adaptability and anti-regression capabilities. Fahmy *et al.* [266] introduced Simulator-based Explanations for DNN Failures (SEDE) as a technique aimed at bolstering the quality

assurance of DNNs within safety-critical systems. SEDE proficiently identifies and simulates events that trigger hazards, leading to DNN failures. This is achieved by generating images with features akin to those causing failures, which are then used for retraining, ultimately improving DNN accuracy.

6.8 Conclusion

To address the issue of high labeling costs for 3D point cloud data, we propose a novel approach called PCPrior, which aims to prioritize test inputs that are likely to be misclassified. By focusing on these challenging inputs, developers can allocate their limited labeling budgets more efficiently, ensuring that the most critical test cases are labeled first, which can lead to cost savings and a more cost-effective testing process. The core idea behind PCPrior is that test inputs closer to the decision boundary of the model are more likely to be predicted incorrectly. In order to capture the spatial relationship between a point cloud test and the decision boundary, we adopt a vectorization approach that transforms the point cloud data into a low-dimensional space, towards revealing the underlying proximity between the point cloud data and the decision boundary indirectly. To implement the vectorization strategy, we generate four distinct types of features for each point cloud (test): Spatial Features, Mutation Features, Prediction Features, and Uncertainty Features. For each test input, the four generated features are concatenated into a final feature vector. Subsequently, PCPrior employs a ranking model to automatically learn the probability of a test input being mispredicted by the model based on its final feature vector. Finally, PCPrior utilized the obtained probability values to rank all the test inputs. In order to assess the performance of PCPrior, we conducted a comprehensive evaluation involving a diverse set of 165 subjects. These subjects encompass both natural datasets and noise datasets. We compared the effectiveness of PCPrior with several established test prioritization approaches that have exhibited effectiveness in prior studies. The empirical results demonstrate the remarkable effectiveness of PCPrior. Specifically, on natural datasets, PCPrior consistently performs better than all the comparative test prioritization approaches, yielding an improvement ranging from 10.99% to 66.94% in terms of APFD. Moreover, on noisy datasets, the improvement ranges from 16.62% to 53%.

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

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

7 Conclusion and Future Work

In this chapter, we provide a conclusion to this dissertation and present promising directions for future research.

Contents

7.1 Conclusion

This dissertation focused on test prioritization for deep neural networks. Specifically, we concentrate on four special test prioritization scenarios: video classification, GNN classification, compressed DNN classification, and 3D shape classification. Below, we provide detailed explanations of the test prioritization methods we proposed for each specific scenario.

In the first part, we proposed VRank, a test prioritization approach specifically designed for video test inputs. The core idea is that test inputs situated closer to the decision boundary of the model are at a higher risk of being predicted incorrectly. To capture the spatial relationship between a video test and the decision boundary, we designed four types of feature generation strategies tailored to video-type tests. Each of these feature types captures essential aspects of the video tests and the model's classification behavior specific to videos. The evaluation results affirmed the effectiveness of VRank on both the natural and noisy datasets.

In the second part, we proposed NodeRank, a GNN-oriented test prioritization approach. NodeRank leverages mutation testing to prioritize graph-structured test inputs. The core idea is that a test is considered more likely to be misclassified if it can kill many mutated models and produce different prediction results with many mutated inputs. The evaluation results demonstrated that NodeRank outperformed all the compared test prioritization approaches on both natural and adversarial test inputs.

In the third part, we proposed PriCod, a test prioritization approach tailored to compressed DNNs. PriCod leverages the behavior disparities caused by model compression, along with the embeddings of test inputs, to effectively prioritize potentially misclassified tests. The core premise is that significant behavior disparities between the models indicate potential misclassifications and that inputs near decision boundaries are more likely to be misclassified. The evaluation results demonstrated the effectiveness of PriCod on natural, noisy, and adversarial test inputs, showing that PriCod outperforms all the compared test prioritization approaches across all three types of scenarios.

In the final part, we proposed PCPrior, a test prioritization method specifically designed for 3D point clouds. PCPrior leverages the unique characteristics of 3D point clouds to prioritize tests. The core idea is that test input close to the decision boundary of the model is more likely to be misclassified. To this end, we carefully designed a group of feature generation strategies tailored to 3D point clouds and utilized the generated features for each test for test prioritization. Our evaluation demonstrated that PCPrior outperformed all the compared test prioritization approaches in both natural and noisy scenarios.

7.2 Future Work

In this section, we discuss some promising future directions.

- **Test prioritization for LLMs with a focus on identifying test inputs that will lead the model to generate low-quality or inaccurate outputs** As the utility of Large Language Models (LLMs) expands across various domains, the demand for robust and reliable models increases. A critical aspect of achieving this is through effective testing. One promising direction is to develop test prioritization approaches to identify and prioritize tests (typically questions) that

will lead the model to output low-quality or inaccurate answers. Identifying and prioritizing such challenging tests can accelerate the LLM debugging process and enhance overall testing efficiency, enabling further optimization of LLMs.

- **Test prioritization for LLMs with a focus on identifying test inputs that will lead the model to generate sensitive, private, or aggressive outputs** Besides generating low-quality or inaccurate outputs, LLMs can inadvertently produce sensitive, private, or aggressive content. Identifying and prioritizing the tests that will lead the model to output such content is crucial in mitigating potential risks. Therefore, another promising direction we focus on is to perform test prioritization from this perspective, with the aim of preventing privacy breaches and the spread of harmful content.
- **Prioritizing speech test inputs** In this dissertation, we have focused on four specific test prioritization scenarios: 3D shape classification, GNN classification, compressed DNN classification, and video classification. There are also some other special scenarios that deserve attention, such as speech classification. Specifically, with the wide adoption of automated speech recognition (ASR) systems, testing and enhancing ASR systems has become increasingly crucial. However, collecting and executing speech test cases is generally costly and time-consuming. Therefore, prioritizing potentially misclassified speech test cases and labeling such inputs first can contribute to quick debugging and enhance testing efficiency.
- **Applying the proposed test prioritization methods to active learning** The aim of test prioritization is to find tests that are more likely to be incorrectly predicted by models. In the field of active learning, theoretically, these tests can also be used to retrain models to enhance their performance. Therefore, a promising research direction is to evaluate whether our proposed test prioritization method can be effectively applied in the field of active learning to improve model performance. This can be achieved by comparing the effectiveness of our proposed method versus existing active learning methods.
- **Expanding to large-scale datasets** In the future, we plan to extend our proposed test prioritization methods to large-scale datasets. This will involve addressing several technical challenges: **1) Increasing complexity of ranking models** In large-scale datasets, simpler ranking models may suffer from underfitting during training. Therefore, it will be necessary to design more complex ranking models that can handle the increased feature dimensionality. **2) Computational resource demands** Large-scale datasets require more computational resources. Our proposed test prioritization approaches, which rely on extracting and processing complex features to train the internal ranking model, could become slow without sufficient computational power. To address this, optimizing the computational efficiency of the feature extraction and model training processes will be critical to ensure scalability.
- **Addressing limitations of the proposed test prioritization approaches.** Our current test prioritization methods face some limitations: **1) Overfitting on small datasets** In scenarios with small datasets, the ranking models in our methods can suffer from overfitting during training, which can lead to a decline in performance. **2) Imbalance in training samples when DNN accuracy is high** When the DNN model has very high accuracy, the number of incorrect predictions in the training data can become very small. This can lead to an imbalance in the training samples, which can affect the ranking model's ability to

learn and classify incorrect predictions. To address these issues, in future work, we will focus on the following improvements: 1) we will adopt data augmentation techniques in small dataset scenarios to generate more relevant samples for training the ranking model, thereby reducing the risk of overfitting. 2) For cases where the DNN model has high accuracy, we will employ oversampling techniques to increase the number of minority class samples in the training set. This will help balance the minority and majority classes, enabling better training of the ranking model.

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:

- **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). Accepted for publication on Jan. 15, 2024.
- **Yinghua Li**, Xueqi Dang, Weiguo Pian, Andrew Habib, Jacques Klein, Tegawendé F. Bissyandé. Test Input Prioritization for Graph Neural Networks. IEEE Transactions on Software Engineering (TSE). Accepted for publication on Mar. 31, 2024.
- **Yinghua Li**, Xueqi Dang, Lei Ma, Jacques Klein, Tegawendé F. Bissyandé. Prioritizing Test Cases for Deep Learning-based Video Classifiers. Empirical Software Engineering (EMSE). Accepted for publication on Jun. 20, 2024.
- **Yinghua Li**, Xueqi Dang, Jacques Klein, Yves Le Traon, Tegawendé F. Bissyandé. PriCod: Prioritizing Test Inputs for Compressed Deep Neural Networks. Under Review in ACM Transactions on Software Engineering and Methodology (TOSEM), 2024.

Papers not included in this dissertation:

- **Yinghua Li**, Xueqi Dang, Haoye Tian, Tiezhu Sun, Zhijie Wang, Lei Ma, Jacques Klein, Tegawendé F. Bissyandé. An Empirical Study of AI Techniques in Mobile Applications. Under Review in Journal of Systems and Software (JSS), 2024.
- 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.

- 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.
- Haoye Tian, **Yinghua Li**, Weiguo Pian, Abdoul Kader Kaboré, Kui Liu, Andrew Habib, Jacques Klein, Tegawendé F. Bissyandé. Predicting Patch Correctness Based on the Similarity of Failing Test Cases. ACM Transactions on Software Engineering and Methodology (TOSEM). Accepted for publication on Jan. 10, 2022.
- Haoye Tian, Kui Liu, **Yinghua Li**, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, Tegawendé F. Bissyandé. The Best of Both Worlds: Combining Learned Embeddings with Engineered Features for Accurate Prediction of Correct Patches. ACM Transactions on Software Engineering and Methodology (TOSEM). Accepted for publication on Nov. 1, 2022.
- Weiguo Pian, **Yinghua Li**, Tiezhu Sun, Yewei Song, Xunzhu Tang, Andrew Habib, Jacques Klein, Tegawendé F. Bissyandé. You Don't Have to Say Where to Edit! Joint Learning to Localize and Edit Source Code. Under Review in ACM Transactions on Software Engineering and Methodology (TOSEM), 2024.

Services

Reviewer:

- ACM Transactions on Software Engineering and Methodology
- IEEE Transactions on Industrial Informatics
- IEEE Transactions on Reliability
- Empirical Software Engineering
- Soft Computing

External Reviewer:

- ISSTA'22, ISSTA'23, ISSTA'24
- ESEC/FSE'22
- ASE'22
- SANER'22

Bibliography

[1] Q. Chen, S. Tang, Q. Yang, and S. Fu, "Cooper: Cooperative perception for connected autonomous vehicles based on 3d point clouds," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 514–524, IEEE, 2019.

[2] M. A. Uy, Q.-H. Pham, B.-S. Hua, T. Nguyen, and S.-K. Yeung, "Revisiting point cloud classification: A new benchmark dataset and classification model on real-world data," in *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 1588–1597, 2019.

[3] X. Huang, X. Cheng, Q. Geng, B. Cao, D. Zhou, P. Wang, Y. Lin, and R. Yang, "The apolloscape dataset for autonomous driving," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pp. 954–960, 2018.

[4] X. Yue, B. Wu, S. A. Seshia, K. Keutzer, and A. L. Sangiovanni-Vincentelli, "A lidar point cloud generator: from a virtual world to autonomous driving," in *Proceedings of the 2018 ACM on International Conference on Multimedia Retrieval*, pp. 458–464, 2018.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] D. Tran, H. Wang, L. Torresani, J. Ray, Y. LeCun, and M. Paluri, "A closer look at spatiotemporal convolutions for action recognition," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pp. 6450–6459, 2018.

[15] 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.

[16] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic graph cnn for learning on point clouds," *ACM Transactions on Graphics (tog)*, vol. 38, no. 5, pp. 1–12, 2019.

[17] W. Wu, Z. Qi, and L. Fuxin, "Pointconv: Deep convolutional networks on 3d point clouds," in *Proceedings of the IEEE/CVF Conference on computer vision and pattern recognition*, pp. 9621–9630, 2019.

[18] Y. Cui, R. Chen, W. Chu, L. Chen, D. Tian, Y. Li, and D. Cao, "Deep learning for image and point cloud fusion in autonomous driving: A review," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 2, pp. 722–739, 2021.

[19] Y. Tian, W. Zhang, M. Wen, S.-C. Cheung, C. Sun, S. Ma, and Y. Jiang, "Finding deviated behaviors of the compressed dnn models for image classifications," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–32, 2023.

[20] 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.

[21] L. Wang, W. Li, W. Li, and L. Van Gool, "Appearance-and-relation networks for video classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1430–1439, 2018.

[22] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning spatiotemporal features with 3d convolutional networks," in *Proceedings of the IEEE international conference on computer vision*, pp. 4489–4497, 2015.

[23] D. Tran, J. Ray, Z. Shou, S.-F. Chang, and M. Paluri, "Convnet architecture search for spatiotemporal feature learning," *arXiv preprint arXiv:1708.05038*, 2017.

[24] C. Feichtenhofer, H. Fan, J. Malik, and K. He, "Slowfast networks for video recognition," in *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 6202–6211, 2019.

[25] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proceedings. 2005 IEEE international joint conference on neural networks*, vol. 2, pp. 729–734, 2005.

[26] 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.

[27] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations*, 2018.

[28] M. Xu, J. Liu, Y. Liu, F. X. Lin, Y. Liu, and X. Liu, "A first look at deep learning apps on smartphones," in *The World Wide Web Conference*, pp. 2125–2136, 2019.

[29] Z. Sun, R. Sun, L. Lu, and A. Mislove, "Mind your weight (s): A large-scale study on insufficient machine learning model protection in mobile apps," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1955–1972, 2021.

[30] T. Choudhary, V. Mishra, A. Goswami, and J. Sarangapani, "A comprehensive survey on model compression and acceleration," *Artificial Intelligence Review*, vol. 53, pp. 5113–5155, 2020.

[31] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," *arXiv preprint arXiv:1702.03044*, 2017.

[32] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang, *et al.*, "Tensorflow lite micro: Embedded machine learning for tinyml systems," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 800–811, 2021.

[33] M. Thakkar and M. Thakkar, "Introduction to core ml framework," *Beginning Machine Learning in iOS: CoreML Framework*, pp. 15–49, 2019.

[34] U. Fadlilah, B. Handaga, *et al.*, "The development of android for indonesian sign language using tensorflow lite and cnn: an initial study," in *Journal of Physics: Conference Series*, vol. 1858, p. 012085, IOP Publishing, 2021.

[35] B. Douillard, J. Underwood, N. Kuntz, V. Vlaskine, A. Quadros, P. Morton, and A. Frenkel, "On the segmentation of 3d lidar point clouds," in *2011 IEEE International Conference on Robotics and Automation*, pp. 2798–2805, IEEE, 2011.

[36] W. Lemkens, P. Kaur, K. Buys, P. Slaets, T. Tuytelaars, and J. De Schutter, "Multi rgb-d camera setup for generating large 3d point clouds," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1092–1099, IEEE, 2013.

[37] P. Achlioptas, O. Diamanti, I. Mitliagkas, and L. Guibas, "Learning representations and generative models for 3d point clouds," in *International conference on machine learning*, pp. 40–49, PMLR, 2018.

[38] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 652–660, 2017.

[39] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao, "3d shapenets: A deep representation for volumetric shapes," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1912–1920, 2015.

[40] A. X. Chang, T. Funkhouser, L. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, *et al.*, "Shapenet: An information-rich 3d model repository," *arXiv preprint arXiv:1512.03012*, 2015.

[41] I. Armeni, O. Sener, A. R. Zamir, H. Jiang, I. Brilakis, M. Fischer, and S. Savarese, "3d semantic parsing of large-scale indoor spaces," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1534–1543, 2016.

[42] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "Pointnet++: Deep hierarchical feature learning on point sets in a metric space," *Advances in neural information processing systems*, vol. 30, 2017.

[43] H. Zheng, J. Chen, and H. Jin, "Certpri: Certifiable prioritization for deep neural networks via movement cost in feature space," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1–13, IEEE, 2023.

[44] Z. Wei, H. Wang, I. Ashraf, and W. Chan, "Predictive mutation analysis of test case prioritization for deep neural networks," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pp. 682–693, IEEE, 2022.

[45] Y. Tao, C. Tao, H. Guo, and B. Li, "Tpfl: Test input prioritization for deep neural networks based on fault localization," in *International Conference on Advanced Data Mining and Applications*, pp. 368–383, Springer, 2022.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] J. Zhou, F. Li, J. Dong, H. Zhang, and D. Hao, "Cost-effective testing of a deep learning model through input reduction," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pp. 289–300, IEEE, 2020.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM conference on recommender systems*, pp. 191–198, 2016.

[63] S. Ghosh, S. J. Sunny, and R. Roney, "Accident detection using convolutional neural networks," in *2019 International Conference on Data Science and Communication (IconDSC)*, pp. 1–6, IEEE, 2019.

[64] A. K. Agrawal, K. Agarwal, J. Choudhary, A. Bhattacharya, S. Tangudu, N. Makhija, and B. Rajitha, "Automatic traffic accident detection system using resnet and svm," in *2020 Fifth International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN)*, pp. 71–76, IEEE, 2020.

[65] S. Bouhsissin, N. Sael, and F. Benabbou, "Enhanced vgg19 model for accident detection and classification from video," in *2021 International Conference on Digital Age & Technological Advances for Sustainable Development (ICDATA)*, pp. 39–46, IEEE, 2021.

[66] L. Peng, H. Wang, and J. Li, "Uncertainty evaluation of object detection algorithms for autonomous vehicles," *Automotive Innovation*, vol. 4, no. 3, pp. 241–252, 2021.

[67] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Testing deep neural networks," *arXiv preprint arXiv:1803.04792*, 2018.

[68] O. Team, "Open source computer vision library," 2023. Accessed May 2023.

[69] D. Zeng, K. Liu, S. Lai, G. Zhou, and J. Zhao, "Relation classification via convolutional deep neural network," in *Proceedings of COLING 2014, the 25th international conference on computational linguistics: technical papers*, pp. 2335–2344, 2014.

[70] 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.

[71] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

[72] 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.

[73] X. Du, X. Xie, Y. Li, L. Ma, Y. Liu, and J. Zhao, "Deepstellar: Model-based quantitative analysis of stateful deep learning systems," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 477–487, 2019.

[74] D. Cheng, C. Cao, C. Xu, and X. Ma, "Manifesting bugs in machine learning code: An explorative study with mutation testing," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 313–324, IEEE, 2018.

[75] A. Aggarwal, P. Lohia, S. Nagar, K. Dey, and D. Saha, "Black box fairness testing of machine learning models," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 625–635, 2019.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] C. Fang, Z. Chen, K. Wu, and Z. Zhao, "Similarity-based test case prioritization using ordered sequences of program entities," *Software Quality Journal*, vol. 22, pp. 335–361, 2014.

[81] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[82] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, 2014.

[83] L. Liberti, C. Lavor, N. Maculan, and A. Mucherino, "Euclidean distance geometry and applications," *SIAM review*, vol. 56, no. 1, pp. 3–69, 2014.

[84] M. Malkauthekar, "Analysis of euclidean distance and manhattan distance measure in face recognition," in *Third International Conference on Computational Intelligence and Information Technology (CIIT 2013)*, pp. 503–507, IET, 2013.

[85] F. Pillichshammer, "On the sum of squared distances in the euclidean plane," *Archiv der Mathematik*, vol. 74, no. 6, pp. 472–480, 2000.

[86] I. Cohen, Y. Huang, J. Chen, J. Benesty, J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," *Noise reduction in speech processing*, pp. 1–4, 2009.

[87] 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.

[88] 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.

[89] 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.

[90] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[91] T. P. Minka, "A comparison of numerical optimizers for logistic regression," *Unpublished draft*, pp. 1–18, 2003.

[92] 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.

[93] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of big data*, vol. 6, no. 1, pp. 1–48, 2019.

[94] L. Perez and J. Wang, "The effectiveness of data augmentation in image classification using deep learning," *arXiv preprint arXiv:1712.04621*, 2017.

[95] A. Mikołajczyk and M. Grochowski, "Data augmentation for improving deep learning in image classification problem," in *2018 international interdisciplinary PhD workshop (IIPhDW)*, pp. 117–122, IEEE, 2018.

[96] L. Taylor and G. Nitschke, "Improving deep learning with generic data augmentation," in *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 1542–1547, IEEE, 2018.

[97] S. Paul, "Video classification with transformers.," 2023. Accessed 10 January 2024.

[98] L. Kezebou, V. Oludare, K. Panetta, J. Intriligator, and S. Agaian, "Highway accident detection and classification from live traffic surveillance cameras: a comprehensive dataset and video action recognition benchmarking," in *Multimodal Image Exploitation and Learning 2022*, vol. 12100, pp. 240–250, SPIE, 2022.

[99] H. Kuehne, H. Jhuang, E. Garrote, T. Poggio, and T. Serre, "Hmdb: a large video database for human motion recognition," in *2011 International conference on computer vision*, pp. 2556–2563, IEEE, 2011.

[100] K. Soomro, A. R. Zamir, and M. Shah, "Ucf101: A dataset of 101 human actions classes from videos in the wild," *arXiv preprint arXiv:1212.0402*, 2012.

[101] 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.

[102] 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.

[103] 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.

[104] T. K. Kim, "T test as a parametric statistic," *Korean journal of anesthesiology*, vol. 68, no. 6, pp. 540–546, 2015.

[105] 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.

[106] K. Kelley and K. J. Preacher, "On effect size.," *Psychological methods*, vol. 17, no. 2, p. 137, 2012.

[107] L. Du, "How much deep learning does neural style transfer really need? an ablation study," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pp. 3150–3159, 2020.

[108] 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.

[109] 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*, jan 2024.

[110] 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.

[111] 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.

[112] L. Ma, F. Zhang, M. Xue, B. Li, Y. Liu, J. Zhao, and Y. Wang, "Combinatorial testing for deep learning systems," *arXiv preprint arXiv:1806.07723*, 2018.

[113] 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.

[114] H. de S. Campos Junior, M. A. P. Araújo, J. M. N. David, R. Braga, F. Campos, and V. Ströele, "Test case prioritization: a systematic review and mapping of the literature," in *Proceedings of the XXXI Brazilian Symposium on Software Engineering*, pp. 34–43, 2017.

[115] Q. Luo, K. Moran, and D. Poshyvanyk, "A large-scale empirical comparison of static and dynamic test case prioritization techniques," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 559–570, 2016.

[116] 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.

[117] F. Wu, A. Souza, T. Zhang, C. Fifty, T. Yu, and K. Weinberger, "Simplifying graph convolutional networks," in *International conference on machine learning*, pp. 6861–6871, PMLR, 2019.

[118] M. Jiang, Z. Li, S. Zhang, S. Wang, X. Wang, Q. Yuan, and Z. Wei, "Drug–target affinity prediction using graph neural network and contact maps," *RSC advances*, vol. 10, no. 35, pp. 20701–20712, 2020.

[119] P. Bongini, M. Bianchini, and F. Scarselli, "Molecular generative graph neural networks for drug discovery," *Neurocomputing*, vol. 450, pp. 242–252, 2021.

[120] 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.

[121] 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.

[122] 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.

[123] 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.

[124] 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.

[125] 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.

[126] I. D. Mienye and Y. Sun, "A survey of ensemble learning: Concepts, algorithms, applications, and prospects," *IEEE Access*, vol. 10, pp. 99129–99149, 2022.

[127] O. Sagi and L. Rokach, "Ensemble learning: A survey," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 4, p. e1249, 2018.

[128] R. Polikar, "Ensemble learning," in *Ensemble machine learning*, pp. 1–34, Springer, 2012.

[129] 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.

[130] T. Loise, X. Devroey, G. Perrouin, M. Papadakis, and P. Heymans, "Towards security-aware mutation testing," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 97–102, IEEE, 2017.

[131] Z.-H. Zhou and Z.-H. Zhou, "Ensemble learning," *Machine learning*, pp. 181–210, 2021.

[132] A. Mohammed and R. Kora, "An effective ensemble deep learning framework for text classification," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 10, pp. 8825–8837, 2022.

[133] P. Goel, R. Jain, A. Nayyar, S. Singhal, and M. Srivastava, "Sarcasm detection using deep learning and ensemble learning," *Multimedia Tools and Applications*, pp. 1–24, 2022.

[134] D. Che, Q. Liu, K. Rasheed, and X. Tao, "Decision tree and ensemble learning algorithms with their applications in bioinformatics," *Software tools and algorithms for biological systems*, pp. 191–199, 2011.

[135] J. Chen, Z. Li, and S. Qin, "Ensemble learning for assessing degree of humor," in *2022 International Conference on Big Data, Information and Computer Network (BDICN)*, pp. 492–498, IEEE, 2022.

[136] F. Divina, A. Gilson, F. Goméz-Vela, M. García Torres, and J. F. Torres, "Stacking ensemble learning for short-term electricity consumption forecasting," *Energies*, vol. 11, no. 4, p. 949, 2018.

[137] N. Littlestone and M. K. Warmuth, "The weighted majority algorithm," *Information and computation*, vol. 108, no. 2, pp. 212–261, 1994.

[138] D. Zügner and S. Günnemann, "Adversarial attacks on graph neural networks via meta learning," *arXiv preprint arXiv:1902.08412*, 2019.

[139] 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.

[140] A. Bojchevski and S. Günnemann, "Adversarial attacks on node embeddings via graph poisoning," in *International Conference on Machine Learning*, pp. 695–704, PMLR, 2019.

[141] Y. Li, W. Jin, H. Xu, and J. Tang, "Deeprobust: A pytorch library for adversarial attacks and defenses," *arXiv preprint arXiv:2005.06149*, 2020.

[142] Z. Liu, Y. Dou, P. S. Yu, Y. Deng, and H. Peng, "Alleviating the inconsistency problem of applying graph neural network to fraud detection," in *Proceedings of the 43rd international ACM SIGIR conference on research and development in information retrieval*, pp. 1569–1572, 2020.

[143] 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.

[144] W. Pian, Y. Wu, X. Qu, J. Cai, and Z. Kou, "Spatial-temporal dynamic graph attention networks for ride-hailing demand prediction," *arXiv preprint arXiv:2006.05905*, 2020.

[145] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[146] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[147] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[148] J. Du, S. Zhang, G. Wu, J. M. Moura, and S. Kar, "Topology adaptive graph convolutional networks," *arXiv preprint arXiv:1710.10370*, 2017.

[149] T.-Y. Liu *et al.*, "Learning to rank for information retrieval," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 3, pp. 225–331, 2009.

[150] R. E. Wright, "Logistic regression.," 1995.

[151] 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.

[152] 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.

[153] 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.

[154] X. Zou, Q. Zheng, Y. Dong, X. Guan, E. Kharlamov, J. Lu, and J. Tang, "Tdgia: Effective injection attacks on graph neural networks," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pp. 2461–2471, 2021.

[155] 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.

[156] 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.

[157] H. Wu, C. Wang, Y. Tyshetskiy, A. Docherty, K. Lu, and L. Zhu, "Adversarial examples on graph data: Deep insights into attack and defense," *arXiv preprint arXiv:1903.01610*, 2019.

[158] 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.

[159] 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.

[160] T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey, "Meta-learning in neural networks: A survey," *IEEE transactions on pattern analysis and machine intelligence*, vol. 44, no. 9, pp. 5149–5169, 2021.

[161] X. Dong, Z. Yu, W. Cao, Y. Shi, and Q. Ma, "A survey on ensemble learning," *Frontiers of Computer Science*, vol. 14, no. 2, pp. 241–258, 2020.

[162] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd international conference on software engineering*, pp. 1–10, 2011.

[163] P. E. McKnight and J. Najab, "Mann-whitney u test," *The Corsini encyclopedia of psychology*, pp. 1–1, 2010.

[164] G. Chryssolouris, M. Lee, and A. Ramsey, "Confidence interval prediction for neural network models," *IEEE Transactions on neural networks*, vol. 7, no. 1, pp. 229–232, 1996.

[165] J. Cohen, "A power primer.," 2016.

[166] 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.

[167] 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.

[168] N. Gökçe, M. Eminov, and F. Belli, "Coverage-based, prioritized testing using neural network clustering," in *Computer and Information Sciences–ISCIS 2006: 21th International Symposium, Istanbul, Turkey, November 1-3, 2006. Proceedings 21*, pp. 1060–1071, Springer, 2006.

[169] F. Belli and C. J. Budnik, "Test minimization for human-computer interaction," *Applied Intelligence*, vol. 26, pp. 161–174, 2007.

[170] N. GÖKÇE, F. Belli, M. EMİNLİ, and B. T. Dincer, "Model-based test case prioritization using cluster analysis: a soft-computing approach," *Turkish Journal of Electrical Engineering and Computer Sciences*, vol. 23, no. 3, pp. 623–640, 2015.

[171] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing android apps," *Information and Software Technology*, vol. 81, pp. 154–168, 2017.

[172] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, "Towards mutation analysis of android apps," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 1–10, IEEE, 2015.

[173] A. Guerriero, R. Pietrantuono, and S. Russo, "Operation is the hardest teacher: estimating dnn accuracy looking for mispredictions," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 348–358, IEEE, 2021.

[174] 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*, 2023.

[175] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," *arXiv preprint arXiv:1802.05668*, 2018.

[176] N. Mairittha, T. Mairittha, and S. Inoue, "On-device deep learning inference for efficient activity data collection," *Sensors*, vol. 19, no. 15, p. 3434, 2019.

[177] N. Mairittha, T. Mairittha, and S. Inoue, "On-device deep personalization for robust activity data collection," *Sensors*, vol. 21, no. 1, p. 41, 2020.

[178] N. Mairittha, T. Mairittha, and S. Inoue, "Improving activity data collection with on-device personalization using fine-tuning," in *Adjunct Proceedings of the 2020 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2020 ACM International Symposium on Wearable Computers*, pp. 255–260, 2020.

[179] H. V. Nguyen and L. Bai, "Cosine similarity metric learning for face verification," in *Asian conference on computer vision*, pp. 709–720, Springer, 2010.

[180] W.-Y. Chiu, G. G. Yen, and T.-K. Juan, "Minimum manhattan distance approach to multiple criteria decision making in multiobjective optimization problems," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 6, pp. 972–985, 2016.

[181] T. Meng, X. Jing, Z. Yan, and W. Pedrycz, "A survey on machine learning for data fusion," *Information Fusion*, vol. 57, pp. 115–129, 2020.

[182] 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.

[183] 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*, 2023.

[184] 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.

[185] A. Panichella, F. M. Kifetew, and P. Tonella, "A large scale empirical comparison of state-of-the-art search-based test case generators," *Information and Software Technology*, vol. 104, pp. 236–256, 2018.

[186] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[187] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[188] T. Klove, T.-T. Lin, S.-C. Tsai, and W.-G. Tzeng, "Permutation arrays under the chebyshev distance," *IEEE Transactions on Information Theory*, vol. 56, no. 6, pp. 2611–2617, 2010.

[189] V. González-Castro, R. Alaiz-Rodríguez, and E. Alegre, "Class distribution estimation based on the hellinger distance," *Information Sciences*, vol. 218, pp. 146–164, 2013.

[190] V. M. Panaretos and Y. Zemel, "Statistical aspects of wasserstein distances," *Annual review of statistics and its application*, vol. 6, pp. 405–431, 2019.

[191] V. Chandrasekaran and P. Shah, "Relative entropy optimization and its applications," *Mathematical Programming*, vol. 161, pp. 1–32, 2017.

[192] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[193] Y. Luo, M. Wang, H. Zhou, Q. Yao, W.-W. Tu, Y. Chen, W. Dai, and Q. Yang, "Autocross: Automatic feature crossing for tabular data in real-world applications," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1936–1945, 2019.

[194] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images," 2009.

[195] S. P. Mohanty, D. P. Hughes, and M. Salathé, "Using deep learning for image-based plant disease detection," *Frontiers in plant science*, vol. 7, p. 1419, 2016.

[196] N. Sharma, V. Jain, and A. Mishra, "An analysis of convolutional neural networks for image classification," *Procedia computer science*, vol. 132, pp. 377–384, 2018.

[197] Zeroshot, "Twitter financial news topic dataset," 2023.

[198] H. Hu, Y. Huang, Q. Chen, T. Y. Zhuo, and C. Chen, "A first look at on-device models in ios apps," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 1, pp. 1–30, 2023.

[199] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang, *et al.*, "Tensorflow lite micro: Embedded machine learning for tinyml systems," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 800–811, 2021.

[200] M. Thakkar, *Beginning machine learning in ios: CoreML framework*. Apress, 2019.

[201] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[202] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[203] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[204] M. Lin, Q. Chen, and S. Yan, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013.

[205] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

[206] Y. Yu, X. Si, C. Hu, and J. Zhang, "A review of recurrent neural networks: Lstm cells and network architectures," *Neural computation*, vol. 31, no. 7, pp. 1235–1270, 2019.

[207] R. Dey and F. M. Salem, "Gate-variants of gated recurrent unit (gru) neural networks," in *2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS)*, pp. 1597–1600, IEEE, 2017.

[208] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[209] T. B. Brown, D. Mané, A. Roy, M. Abadi, and J. Gilmer, "Adversarial patch," *arXiv preprint arXiv:1712.09665*, 2017.

[210] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *Artificial intelligence safety and security*, pp. 99–112, Chapman and Hall/CRC, 2018.

[211] 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, 2022.

[212] 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.

[213] H. Al-Qadasi, Y. Falcone, and S. Bensalem, "Difficulty and severity-oriented metrics for test prioritization in deep learning systems," in *2023 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pp. 40–48, IEEE, 2023.

[214] N. Alshahwan, M. Harman, and A. Marginean, "Software testing research challenges: An industrial perspective," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–10, IEEE, 2023.

[215] 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.

[216] 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.

[217] 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.

[218] 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.

[219] 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.

[220] 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.

[221] 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.

[222] 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.

[223] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," *arXiv preprint arXiv:2002.12543*, 2020.

[224] Z. Wu, Z. Wang, J. Chen, H. You, M. Yan, and L. Wang, "Stratified random sampling for neural network test input selection," *Information and Software Technology*, vol. 165, p. 107331, 2024.

[225] Y. Hao, Z. Huang, H. Guo, and G. Shen, "Test input selection for deep neural network enhancement based on multiple-objective optimization," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 534–545, IEEE, 2023.

[226] X. Wu, J. Shen, W. Zheng, L. Lin, Y. Sui, and A. O. A. Semasaba, "Rnntcs: A test case selection method for recurrent neural networks," *Knowledge-Based Systems*, vol. 279, p. 110955, 2023.

[227] Z. Liu, Y. Feng, Y. Yin, and Z. Chen, "Deepstate: Selecting test suites to enhance the robustness of recurrent neural networks. in 2022 ieee/acm 44th international conference on software engineering (icse). 598–609," *Google Scholar Google Scholar Digital Library Digital Library*, 2022.

[228] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," in *International Conference on Machine Learning*, pp. 4901–4911, PMLR, 2019.

[229] A. H. Yahmed, H. B. Braiek, F. Khomh, S. Bouzidi, and R. Zaatour, "Diverget: a search-based software testing approach for deep neural network quantization assessment," *Empirical Software Engineering*, vol. 27, no. 7, p. 193, 2022.

[230] X. Xie, L. Ma, H. Wang, Y. Li, Y. Liu, and X. Li, "Diffchaser: Detecting disagreements for deep neural networks," International Joint Conferences on Artificial Intelligence Organization, 2019.

[231] B. Mahmood, S. Han, and D.-E. Lee, "Bim-based registration and localization of 3d point clouds of indoor scenes using geometric features for augmented reality," *Remote Sensing*, vol. 12, no. 14, p. 2302, 2020.

[232] B. Mahmood and S. Han, "3d registration of indoor point clouds for augmented reality," in *ASCE International Conference on Computing in Civil Engineering 2019*, pp. 1–8, American Society of Civil Engineers Reston, VA, 2019.

[233] D. Borrmann, A. Nuechter, and T. Wiemann, "Large-scale 3d point cloud processing for mixed and augmented reality," in *2018 IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)*, pp. xxxv–xxxv, IEEE, 2018.

[234] S. Ortega, J. M. Santana, J. Wendel, A. Trujillo, and S. M. Murshed, "Generating 3d city models from open lidar point clouds: Advancing towards smart city applications," *Open Source Geospatial Science for Urban Studies: The Value of Open Geospatial Data*, pp. 97–116, 2021.

[235] C.-Y. Chiu, M. Thelwell, T. Senior, S. Choppin, J. Hart, and J. Wheat, "Comparison of depth cameras for three-dimensional reconstruction in medicine," *Proceedings of the Institution of Mechanical Engineers, Part H: Journal of Engineering in Medicine*, vol. 233, no. 9, pp. 938–947, 2019.

[236] J. Shao, W. Zhang, N. Mellado, P. Grussenmeyer, R. Li, Y. Chen, P. Wan, X. Zhang, and S. Cai, "Automated markerless registration of point clouds from tls and structured light scanner for heritage documentation," *Journal of Cultural Heritage*, vol. 35, pp. 16–24, 2019.

[237] Y. Wang and J. M. Solomon, "Deep closest point: Learning representations for point cloud registration," in *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 3523–3532, 2019.

[238] Q. Wang and M.-K. Kim, "Applications of 3d point cloud data in the construction industry: A fifteen-year review from 2004 to 2018," *Advanced Engineering Informatics*, vol. 39, pp. 306–319, 2019.

[239] M. Simony, S. Milzy, K. Amendey, and H.-M. Gross, "Complex-yolo: An euler-region-proposal for real-time 3d object detection on point clouds," in *Proceedings of the European conference on computer vision (ECCV) workshops*, pp. 0–0, 2018.

[240] A. Ioannidou, E. Chatzilari, S. Nikolopoulos, and I. Kompatsiaris, "Deep learning advances in computer vision with 3d data: A survey," *ACM computing surveys (CSUR)*, vol. 50, no. 2, pp. 1–38, 2017.

[241] A. Kendall and Y. Gal, "What uncertainties do we need in bayesian deep learning for computer vision?," *Advances in neural information processing systems*, vol. 30, 2017.

[242] J. Zhang, X. Zhao, Z. Chen, and Z. Lu, "A review of deep learning-based semantic segmentation for point cloud," *IEEE access*, vol. 7, pp. 179118–179133, 2019.

[243] S. Chen, B. Liu, C. Feng, C. Vallespi-Gonzalez, and C. Wellington, "3d point cloud processing and learning for autonomous driving: Impacting map creation, localization, and perception," *IEEE Signal Processing Magazine*, vol. 38, no. 1, pp. 68–86, 2020.

[244] F. Pomerleau, F. Colas, R. Siegwart, *et al.*, "A review of point cloud registration algorithms for mobile robotics," *Foundations and Trends® in Robotics*, vol. 4, no. 1, pp. 1–104, 2015.

[245] Z. Zhang, Y. Dai, and J. Sun, "Deep learning based point cloud registration: an overview," *Virtual Reality & Intelligent Hardware*, vol. 2, no. 3, pp. 222–246, 2020.

[246] S. A. Bello, S. Yu, C. Wang, J. M. Adam, and J. Li, "Deep learning on 3d point clouds," *Remote Sensing*, vol. 12, no. 11, p. 1729, 2020.

[247] M. G. Larson, "Analysis of variance," *Circulation*, vol. 117, no. 1, pp. 115–121, 2008.

[248] S. Basu and A. DasGupta, "The mean, median, and mode of unimodal distributions: a characterization," *Theory of Probability & Its Applications*, vol. 41, no. 2, pp. 210–223, 1997.

[249] Y. Qian, P. Cao, W. Yin, F. Dai, F. Hu, Z. Yan, *et al.*, "Calculation method of surface shape feature of rice seed based on point cloud," *Computers and Electronics in Agriculture*, vol. 142, pp. 416–423, 2017.

[250] S. Kokoska and D. Zwillinger, *CRC standard probability and statistics tables and formulae*. Crc Press, 2000.

[251] S. Sperandei, "Understanding logistic regression analysis," *Biochemia medica*, vol. 24, no. 1, pp. 12–18, 2014.

[252] M. Prince, "Does active learning work? a review of the research," *Journal of engineering education*, vol. 93, no. 3, pp. 223–231, 2004.

[253] K. M. Al-Gethami, M. T. Al-Akhras, and M. Alawairdhi, "Empirical evaluation of noise influence on supervised machine learning algorithms using intrusion detection datasets," *Security and Communication Networks*, vol. 2021, pp. 1–28, 2021.

[254] Y. Guo, H. Wang, Q. Hu, H. Liu, L. Liu, and M. Bennamoun, "Deep learning for 3d point clouds: A survey," *IEEE transactions on pattern analysis and machine intelligence*, vol. 43, no. 12, pp. 4338–4364, 2020.

[255] X. Liu, M. Yan, and J. Bohg, "Meteornet: Deep learning on dynamic 3d point cloud sequences," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 9246–9255, 2019.

[256] T. Huang and Y. Liu, "3d point cloud geometry compression on deep learning," in *Proceedings of the 27th ACM international conference on multimedia*, pp. 890–898, 2019.

[257] 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.

[258] 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.

[259] Q. Hu, Y. Guo, M. Cordy, X. Xie, L. Ma, M. Papadakis, and Y. Le Traon, "An empirical study on data distribution-aware test selection for deep learning enhancement," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–30, 2022.

[260] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," in *Proceedings of the 17th international conference on Software engineering*, pp. 41–50, 1995.

[261] J. Chen, "Learning to accelerate compiler testing," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pp. 472–475, 2018.

[262] S. Gerasimou, H. F. Eniser, A. Sen, and A. Cakan, "Importance-driven deep learning system testing," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 702–713, 2020.

[263] 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.

[264] 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.

[265] M. Biagiola and P. Tonella, "Testing the plasticity of reinforcement learning-based systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–46, 2022.

[266] H. Fahmy, F. Pastore, L. Briand, and T. Stifter, "Simulator-based explanation and debugging of hazard-triggering events in dnn-based safety-critical systems," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–47, 2023.