Test input prioritization for Machine Learning Classifiers

Xueqi Dang, Yinghua Li, Mike Papadakis, Jacques Klein, Tegawendé F. Bissyandé and Yves Le Traon

Abstract—Machine learning has achieved remarkable success across diverse domains. Nevertheless, concerns about interpretability in black-box models, especially within Deep Neural Networks (DNNs), have become pronounced in safety-critical fields like healthcare and finance. Classical machine learning (ML) classifiers, known for their higher interpretability, are preferred in these domains. Similar to DNNs, classical ML classifiers can exhibit bugs that could lead to severe consequences in practice. Test input prioritization has emerged as a promising approach to ensure the quality of an ML system, which prioritizes potentially misclassified tests so that such tests can be identified earlier with limited manual labeling costs. However, when applying to classical ML classifiers, existing DNN test prioritization methods are constrained from three perspectives: 1) Coverage-based methods are inefficient and time-consuming; 2) Mutation-based methods cannot be adapted to classical ML models due to mismatched model mutation rules; 3) Confidence-based methods are restricted to a single dimension when applying to binary ML classifiers, solely depending on the model's prediction probability for one class. To overcome the challenges, we propose MLPrior, a test prioritization approach specifically tailored for classical ML models. MLPrior leverages the characteristics of classical ML classifiers (i.e., interpretable models and carefully engineered attribute features) to prioritize test inputs. The foundational principles are: 1) tests more sensitive to mutations are more likely to be misclassified, and 2) tests closer to the model's decision boundary are more likely to be misclassified. Building on the first concept, we design mutation rules to generate two types of mutation features (i.e., **model mutation features** and **input mutation features**) for each test. Drawing from the second notion, MLPrior generates **attribute features** of each test based on its attribute values, which can indirectly reveal the proximity between the test and the decision boundary. For each test, MLPrior combines all three types of features of it into a final vector. Subsequently, MLPrior employs a pre-trained ranking model to predict the misclassification probability of each test based on its final vector and ranks tests accordingly. We conducted an extensive study to evaluate MLPrior based on 185 subjects, encompassing natural datasets, mixed noisy datasets, and fairness datasets. The results demonstrate that MLPrior outperforms all the compared test prioritization approaches, with an average improvement of 14.74%∼66.93% on natural datasets, 18.55%∼67.73% on mixed noisy datasets, and 15.34%∼62.72% on fairness datasets.

Index Terms—Test Input Prioritization, Machine Learning, Mutation analysis, Learning to Rank, Labelling

1 Introduction

Machine learning classifiers have seen remarkable success in various domains [1], including image recognition [2], natural language processing [3], [4], and recommendation systems [5], [6]. However, the prevalence of black-box models, especially in deep learning, has raised concerns about their lack of interpretability, which refers to the extent to which a model's internal mechanism and decision-making processes can be comprehended and explained transparently to humans. Interpretability becomes particularly vital in safety-critical domains like healthcare and finance [7], where model decisions can profoundly impact individuals' lives and societal well-being.

Compared to black-box models, classical machine learning (ML) algorithms (e.g., XGBoost [8], decision tree [9] and logistic regression [10]) offer more interpretable solutions, making them an appealing choice for domains that prioritize transparency and comprehensibility.

- *X. Dang, Y. Li, M. Papadakis, J. Klein and T. Bissyandé and Y. Traon are with University of Luxembourg.*
  E-mail: xueqi.dang@uni.lu, yinghua.li@uni.lu, michail.papadakis@uni.lu, jacques.klein@uni.lu, tegawende.bissyande@uni.lu, yves.letraon@uni.lu
- *Xueqi Dang and Yinghua Li are co-first authors. Yinghua Li is the corresponding author.*

While classical ML classifiers are inherently interpretable, ensuring their accuracy and reliability remains a challenge. Testing is a fundamental practice for ensuring the quality of ML systems. However, a significant challenge in ML testing is the labeling cost issue [11] (i.e., labeling test inputs to verify the correctness of predictions can be costly). This challenge arises due to several factors: 1) manual annotation is still the mainstream for labeling; 2) test sets can be large-scale, which increases labeling efforts; 3) domain-specific knowledge can be required in certain domains for labeling tabular data, such as the medical domain [12], [13], [14]. For instance, when applying XGBoost for chronic kidney disease (CKD) detection [12], labelling the CKD dataset for model training/testing requires specialized medical expertise to determine whether a patient has CKD.

To deal with the labelling cost problem, one intuitive solution is to prioritize tests that can cause the ML model to behave incorrectly (i.e., inputs that are more likely to be misclassified by the model). Early identification and labelling of such tests can save the manual labelling effort and enhance the overall efficiency of the testing process. In the literature, various test prioritization approaches [15], [16] have been proposed in the field of DNN testing. These techniques can be broadly classified into three categories: coverage-based [17], [18], [19], confidence-based [16], [20]

and mutation-based [15] approaches.

Coverage-based approaches prioritize test inputs based on the neuron coverage of DNNs. Confidence-based methods identify possibly-misclassified test inputs by quantifying the classifier's output confidence for each test. One notable confidence-based approach is DeepGini [16], which leverages the Gini score as a metric to quantify confidence levels for effective test prioritization. Recently, Weiss *et al.* [20] conducted a comprehensive study to assess existing test prioritization methods, containing the evaluation of a series of confidence-based metrics, including Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy. Mutation-based techniques propose a set of mutation operations and utilize the mutated results for test prioritization. While these approaches have made considerable progress in prioritizing potentially-misclassified test inputs, they still face certain challenges and limitations.

First, prior studies [16] have demonstrated that coverage-based methods are ineffective and time-costly compared to confidence-based approaches. Second, the mutation-based test prioritization approach, PRIMA [15], is not applicable to classical ML models due to the lack of adapted model mutation operators. Third, while confidence-based test prioritization approaches can be adapted for classical ML models, there are several limitations associated with their application in this context. We outline the main limitations as follows. Specific details can be found in the background section (cf. Section 2).

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

In this paper, we propose MLPrior (Classical **ML**-oriented Test **Prior**itization), a test prioritization approach specifically tailored for classical machine learning (ML) models. MLPrior addresses the aforementioned limitations, leveraging the characteristics of classical ML classifiers (i.e., interpretable models and carefully engineered attribute features) to prioritize test inputs. The core ideas behind MLPrior are twofold: 1) tests more sensitive to the injected mutations are more likely to reveal bugs, and 2) test inputs closer to the decision boundary of the model are more likely to be predicted incorrectly. Both premises have been validated by existing studies [21], [22], [23], [24], with a detailed explanation provided in the Background section. Building upon the aforementioned premise, MLPrior utilizes the characteristics of classical ML classifiers to prioritize test inputs, addressing the limitations of confidence-based methods in the following way.

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

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

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

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

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

MLPrior demonstrates broad applicability across various contexts. One specific application pertains to banking loan operations, where classical ML models are employed to determine whether a loan can be granted to a user. In this particular scenario, classical ML models utilize a set of user attributes (e.g., gender, age, and transaction history) to predict the viability of granting a loan to a user. Incorrect predictions can lead to significant losses for the bank. For instance, if the bank mistakenly grants a loan to a user without the ability to repay, these users can fail to meet their repayment obligations, increasing the risk of default and causing damage to the bank's assets. In this context, MLPrior can identify and prioritize users who are more likely to be misclassified by the model. Consequently, two main advantages arise: Firstly, these potentially misclassified users can be prioritized for manual inspection, resulting in a decrease in losses caused by inaccurate predictions generated by the model. Secondly, developers can manually inspect the attributes of misclassified users and analyze which attributes led to prediction errors, using this information to optimize the model.

We conducted an extensive study to evaluate MLPrior's performance utilizing 185 subjects (i.e., paired datasets and ML models). The evaluation encompassed different types of test inputs, including natural data, mixed noisy data, and fairness data. Ensuring fairness in machine learning is essential to prevent bias and discrimination against specific groups during predictions. Fairness has become a critical ethical consideration in diverse machine learning domains, including recruitment, loan approvals, and medical diagnosis [25]. In these domains, the absence of fairness can lead to unjust treatment of particular groups, affecting individuals' lives and rights. Therefore, the evaluation of MLPrior's effectiveness on fairness datasets assumes crucial importance. To generate the fairness datasets, we followed the approach of prior research [26]. Specifically, we selected a group of test inputs and modified their gender and age attribute values while retaining their original labels. Moreover, we carefully selected a group of test prioritization approaches that can be adapted to prioritize test inputs in the context of classical ML models as the comparative methods, which have been demonstrated effective in existing studies [20], [16]. Additionally, we utilize random selection as the baseline approach.

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

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

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

2 Background

2.1 Machine Learning and ML testing

Machine Learning (ML) has gained widespread adoption in various domains, demonstrating significant utility in safety-critical sectors like autonomous vehicle systems [27] and medical intervention protocols [28]. Existing literature [11] pointed out that ML can be broadly classified into two primary branches: classical Machine Learning [29], [8] and Deep Learning [30], [31]. Classical Machine Learning encompasses a range of approaches, including decision trees [9] and logistic regression [10]. These classical algorithms remain widely employed in various industrial applications [32], [33]. DNNs consist of interconnected nodes (neurons) organized in layers, with each layer responsible for learning and abstracting different levels of features from input data. In contrast to DNNs, classical ML models are generally more interpretable [34]. Interpretability in machine learning refers to the degree to which a model's internal mechanisms and decision-making processes can be understood and transparently explained to humans. Interpretability is crucial in domains where transparency and interpretability are essential, such as healthcare [35] and finance [36]. Therefore, classical machine learning models retain distinct advantages in certain application domains.

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

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

1) Risk Management Challenges in Finance Weber *et al.* [37] highlighted that, in the financial field, a high degree of transparency and interpretability is required for effective risk management. The lack of interpretability in black-box models can make it challenging for financial institutions to understand how decisions are made, thereby increasing the difficulty of risk management.

2) Legal and Ethical Issues in Finance Chen *et al.* [38] pointed out that, according to legal and ethical principles, financial companies are required to provide clear explanations for the reasons behind specific loan application rejections. However, with black-box models, loan applicants are unaware of how their scores are calculated. Even if model explanations are provided, there can be a disconnect between the explanations for loan rejection and the actual model calculations, as the explanations could be created after the fact.

3) Trust Issues in Healthcare Adadi *et al.* [39] discussed the constrained acceptance of black-box models in clinical settings due to trust and transparency issues. Moreover, Verdicchio *et al.* [40] raised a vital question: "If doctors cannot understand why a black-box model diagnoses, why should patients trust the treatment recommendations?". This implies that black-box models lack interpretability, making it difficult to explain the fundamental reasons behind their diagnostic or treatment recommendations. Therefore, patients and doctors can be skeptical of the system's suggestions and even refuse to follow its recommendations because they cannot be certain if these recommendations are based on sound medical reasoning. This lack of trust and understanding can significantly affect patients' confidence in the proposed treatments, potentially hindering their willingness to undergo specific medical procedures.

4) Responsibility Issues in Healthcare Smith *et al.* [41] pointed out that if patients are harmed due to recommendations from an opaque AI system (AIS) adopted by clinicians, questions arise about how responsibility will be assigned. Specifically, in the healthcare field, doctors are expected to take responsibility for their decisions. If a black-box system provides incorrect recommendations, doctors will find it challenging to explain why they followed the system's advice, potentially raising legal and ethical liability concerns.

Based on the existing studies [42], [43], [44], in the following, we provide the quantification of the loss resulting from the lack of interpretability in black-box models. Specifically, we employ descriptive terms to quantify the degree of loss in two specific scenarios: medical and financial.

- **Medical Scenario** Amann *et al.* [42] pointed out that, in the medical domain, the lack of interpretability in black-box models can lead to serious legal and ethical uncertainty. Without adequate consideration of interpretability, these technologies can neglect regulatory issues and result in significant harm. Moreover, Grote *et al.* [43] pointed out that in the face of a black-box model lacking interpretability, its clinical decision support can constrain the capabilities of physicians. Specifically, physicians can rigidly adhere to the output of the black-box model to avoid being held accountable. This situation poses a serious threat to the autonomy of physicians.
- **Financial Scenario** Yan *et al.* [44] pointed out that, in the financial domain, the lack of interpretability in the decision mechanisms of black-box models poses a challenge for financial practitioners and regulatory authorities in understanding the factors influencing the model's decisions. This can significantly impact the fairness of loan decisions, potentially resulting in substantial financial losses.

Although interpretability is a valuable trait, it is not the sole factor taken into account when deploying models, especially in the healthcare industry [45], [46]. Deep learning has also demonstrated remarkable success in healthcare applications [47]. However, there are compelling reasons that test prioritization for classical models remains highly necessary.

- **Applicability to Structured Medical Data:** Deep learning finds extensive use in the field of medical imaging [46], aiding in the automatic detection of diseases and tumors. However, a substantial portion of data in the healthcare sector exists in structured tabular formats. Classical machine learning models have demonstrated superior performance when dealing with structured medical data, outperforming deep learning methods [48], [49]. For instance, Shwartz *et al.* [48] pointed out that when handling tabular datasets, the classical ML technique XGBoost outperforms the evaluated DL models.
- **Need for Interpretability:** In healthcare [40], when clinicians need to justify their decisions to patients, having an understanding of the reasoning behind model predictions is essential. Classical machine learning models can provide this crucial information [50].
- **Regulatory Approvals:** Regulatory bodies can require models to elucidate the decision-making processes of a model to facilitate comprehensive treatment risk assessment [7]. The interpretability that classical ML models can provide is crucial for obtaining regulatory approvals.

Machine learning testing involves systematically evaluating and validating machine learning models to ensure their accuracy, reliability, and effectiveness in prediction or decision-making [51], [52], [7], [53]. The primary goal is to reveal disparities between intended and actual behaviors exhibited by ML systems [11]. Compared to traditional software systems, machine learning testing presents distinct challenges. One pivotal challenge is the Oracle Problem [54], which pertains to the difficulty in acquiring accurate labels or ground truth for training and testing data. In the context of testing ML-based systems, automated testing oracles are typically unavailable. Therefore, manual labeling remains the mainstream method, which can lead to substantial labeling costs. In the literature, numerous fields are dedicated to addressing labeling cost concerns, such as test selection [55], [56] and test prioritization [16], [15]. In our study, we concentrate on test prioritization, which will be further elaborated in the subsequent section.

2.2 Test Case Prioritization

In the field of traditional software testing, test case prioritization aims to determine the sequence in which test

cases are executed to uncover defects more effectively. In the literature, numerous techniques for test prioritization have been proposed. The majority of these approaches are rooted in code coverage analysis. Notably, two primary coverage-based techniques are: Coverage-Total Method (CTM) and Coverage-Additional Method (CAM) [57]. CTM operates by sequentially selecting tests with the highest coverage rates, followed by those with progressively lower rates. In cases where tests share the same coverage rate, the method introduces randomness to determine the prioritization. In contrast, CAM distinguishes itself from CTM by its approach. It strategically utilizes feedback from previous selections, iteratively opting for tests that target previously uncovered code structures, thereby incrementally expanding the coverage.

Test input prioritization in the field of Deep Neural Networks (DNNs) [15], [20], [16], [58] aims to enhance the efficiency of testing by focusing on test inputs that are more likely to expose model misclassifications, thereby revealing potential bugs earlier. This approach ensures that crucial test inputs are identified and labeled promptly within the constraints of limited time. Previous research [16] has indicated that confidence-based approaches outperform the aforementioned coverage-based methods. These confidence-based approaches prioritize tests based on the model's confidence. One notable approach is DeepGini [16], which surpasses all existing coverage-based prioritization methods in terms of both effectiveness and efficiency. A recent comprehensive investigation conducted by Weiss *et al.* delved into the capabilities of various confidence-based DNN test input prioritization techniques, such as Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy. They demonstrated the effectiveness of these approaches in identifying potentially misclassified test inputs.

However, while confidence-based test prioritization methods have been proven effective [16] and can be adapted for classical ML models, their application in the context of test prioritization for classical ML models is hindered by several limitations. We discuss these limitations as follows.

- **Single dimension on binary classification models** Binary classification models [59], [60] categorize test inputs into two distinct classes, which limits the application of confidence-based test prioritization approaches to a single dimension. Specifically, when applying confidence-based approaches to these models, the first step is calculating the probabilities for each classification, denoted as $(p, 1-p)$. If the model's prediction probability for a test is $(0.5, 0.5)$, it means the model is most uncertain about this test [16], indicating this test is more likely to be misclassified. The closer a test's p value is to 0.5, the more uncertain the model is about that particular test. Consequently, uncertainty is solely determined by p.
  Regardless of the specific confidence-based test prioritization method employed, tests with p values closer to 0.5 will be prioritized over others. To illustrate this point, consider a hypothetical test set with three tests, and the model's probability vectors for these tests are as follows: t_1 $(0.9, 0.1)$, t_2 $(0.7, 0.3)$, t_3 $(0.8, 0.2)$. Irrespective of the chosen confidence-based test prioritization method, the resulting ranking will be $t_2 \rightarrow t_3 \rightarrow t_1$ because t_2 has the p value (0.7) closest to 0.5, followed by t_3 (p = 0.8), while t_1 has the farthest p value from 0.5 (p = 0.9).
  The above conclusions have been confirmed through our experimental results. For each subject, all confidence-based methods yield identical effectiveness, indicating they produce the same ranking for a given test set.
- **Lack of model-specific insights** Confidence-based approaches for test prioritization consider the model a black box and rely solely on its prediction probability vectors. This neglects the transparency and interpretability of classical ML models, which are mostly white-box and have an understandable decision-making process. As a result, confidence-based approaches fail to incorporate crucial model-specific insights from classical ML models, leading to suboptimal test prioritization.
- **Ignoring attribute features** Furthermore, confidence-based approaches ignore a crucial aspect of the test datasets for classical ML models, namely, the attribute features. These features are carefully engineered by domain experts to effectively capture and represent crucial aspects of the underlying data. They can directly reflect the attribute information of each test input. However, confidence-based approaches ignore this crucial feature information in the process of test prioritization.

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

The first premise is grounded in the well-established practice of traditional mutation testing [21], [22], [23], [61], [62], which considers that test cases sensitive to mutations (able to capture mutants) have a higher capability to detect bugs in software. The second premise has been identified and demonstrated in prior work [24].

2.3 Mutation Testing

Mutation testing [63], [64] is a systematic software testing technique that has gained significant attention in both academic and industrial research communities [65], [66]. The fundamental principle is to introduce small and intentional modifications, called mutants, into the source code of a software system [67]. These mutations simulate potential faults that may occur during the execution of the program. A well-designed test suite should be able to detect the presence of these mutants, indicating its capability to detect real faults in the code [68]. In the context of mutation testing, the term "kill" refers to the ability of a test case to detect a specific mutant [69]. When a test case "kills" a mutant, it means that the test case is able to reveal a difference in behavior between the original program and the mutated version of the program. A test suite with a high "mutation kill" rate is considered more effective and reliable, as it demonstrates a greater ability to detect potential faults or deviations from the expected behavior.

2.4 Automated Labeling Approaches for Machine Learning

Data labeling is a labor-intensive task that is indispensable in the development of supervised machine learning systems [70]. Conventional data labeling methods typically rely on manual effort, which is a time-consuming and costly process. Moreover, in specialized fields like medicine and finance, manual labeling necessitates domain-specific expertise, further increasing its cost. In recent years, various automated or semi-automated data labeling methods [71], [72] have emerged, aimed at reducing the burden of manual labeling and improving the overall labeling efficiency.

Desmond *et al.* [72] introduced a semi-automated data labeling system that views the labeling task as a collaborative effort between human annotators and machine annotators, which are implemented as predictive models. The core of this approach involves a human-machine coactive process facilitated by a semi-supervised predictive model and an active learning selector. In each iteration, the active learning selector prioritizes the most uncertain examples for annotation by human annotators based on the model's predictions. The consistency between human decisions and machine predictions is continuously monitored and presented at various checkpoints, allowing annotators to assess the machine's performance in the labeling task. Once annotators are satisfied with the machine's performance, they can delegate the remaining labeling tasks to the machine (automatic labeling).

Wu *et al.* [73] proposed a semi-automated labeling method based on active learning and label informativeness. Specifically, the SLMAL algorithm selects the most informative example-label pairs for annotation by combining the uncertainty of examples and the informativeness of labels. During this process, the algorithm initially identifies and prioritizes the example-label pairs in need of labeling the most and subsequently employs the nearest neighbors of these highly uncertain pairs to predict their partial labels.

However, semi-automatic labeling comes with several limitations:

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

However, despite the presence of semi-supervised learning, in order to labeling tests more accurate and of higher quality, manual labeling is still the mainstream in the industry [71].

Automated labeling methods offer a potential solution to the aforementioned limitations. Nevertheless, due to the constraints outlined below, fewer automated labeling methods are specifically designed for classical machine learning. To our best knowledge, the known method applicable to labeling for classical machine learning models is Programmatic Labeling [74]. Programmatic labeling automates the labeling process through scripts and programming algorithms, significantly improving the efficiency of data preparation. However, Programmatic Labeling typically requires specialized programming skills to create labeling rules, which may pose a barrier for researchers without a technical background.

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

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

3 Approach

3.1 Overview

In this paper, we propose MLPrior, a test prioritization approach specifically designed for classical ML models. Figure 1 illustrates the workflow of MLPrior. Given a test set T and an ML model M, MLPrior produces a sorted test set T', where test cases that are more likely to be mispredicted by the model are placed at the front. We outline the steps of MLPrior as follows.

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

❷ **Mutation feature generation (model):** Based on the model mutation rules described in Section 3.2, MLPrior generates a set of mutated models for the original ML model M. For each test $t \in T$, MLPrior identifies whether t "kills" each of the mutated models (i.e., whether the predictions made by the mutated model and the original ML model for t are different). This process allows MLPrior to construct a model mutation feature vector, denoted

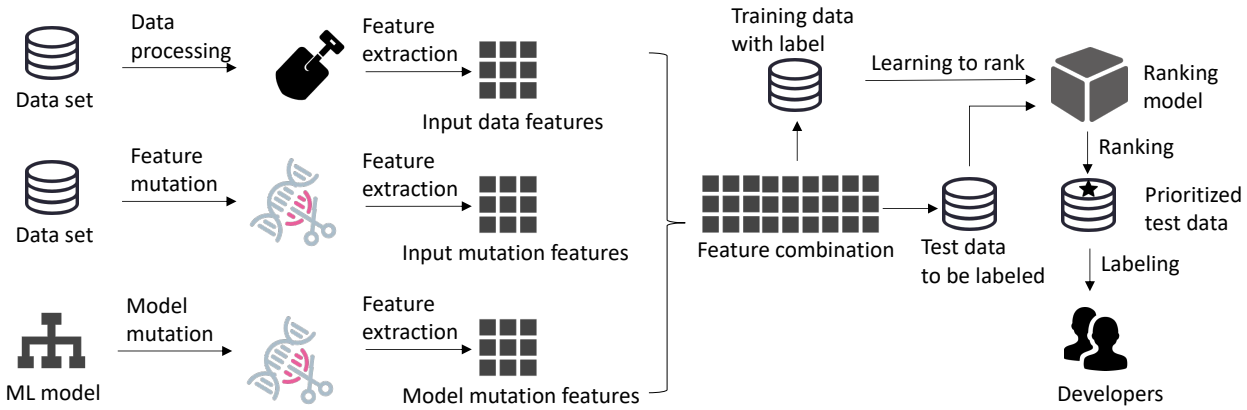

Fig. 1: Overview of MLPrior

as V_t^M. Each element of V_t^M corresponds to a specific mutated model. More specifically, MLPrior sets the i_{th} element of t's model mutation vector to 1 if t kills the i_{th} mutated model. Otherwise, the element is set to 0.

❸ **Mutation feature geneation (inputs):** Based on the input mutation rules outlined in Section 3.2, MLPrior generates mutated inputs for each test instance $t \in T$. By comparing the predictions of model M on the i_{th} mutated input with its predictions on the original test input t, MLPrior constructs an input mutation vector denoted as V_t^I. If the prediction of model M for the i_{th} mutated input is different from that of the original test input t, the i_{th} element of V_t^I is set to 1. Otherwise, it is set to 0.

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

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

In MLPrior, the concept of **feature** is crucial. To demonstrate the processes of feature extraction, combination, and concatenation more intuitively, we provide a typical example. In this example, we delve into the specifics of how MLPrior generates features for a given test t, illustrating each step of the process in detail. Furthermore, we visually illustrated this example in Figure 2 to enhance the presentation of MLPrior's feature generation process.

- **Feeding Attributes of t to MLPrior** Given a classical ML model M and its corresponding test set T, let t be a test instance from the test set T. Given that the dataset for the classical ML model is in a tabular format, we assume the attribute features of t as $t = (s_1, s_2, \ldots, s_n)$. Here, s_n includes both numeric and non-numeric formats (such as strings). In this step, we input the attributes of the test t into MLPrior.
- **Generation of Original Attribute Features** We input the attribute vector of test t, which is $(s_1, s_2, \ldots, s_n)$, into MLPrior. MLPrior then converts all non-numeric attributes into numeric format to construct the original attribute vector of t, represented as $(i_1, i_2, \ldots, i_n)$.
- **Generation of Input Mutation Features** Subsequently, MLPrior generates N mutanted inputs of the test t, denoted as $(t_1, t_2, \ldots, t_N)$. MLPrior then feeds these mutated inputs into the original ML model to make predictions. If the model output for t_i differs from the result of the original sample t, the i_{th} element of the input mutation feature vector will be set to 1; otherwise, it will be set to 0. In this manner, we obtain the input mutation feature vector for t, represented as $(0, 1, \ldots, 0)$. This vector indicates that for the first mutant of t, denoted as t_1, the model's prediction is the same as for t. For the second mutant of t, denoted as t_2, the model's prediction differs from that for t.
- **Generation of Model Mutation Features** For the original model M, MLPrior generates K mutated models, denoted as $(m_1, m_2, \ldots, m_K)$, and inputs the original sample t into these mutated models for prediction. If the prediction of the i_{th} mutated model for t differs from the prediction of the original model M for t, then the i_{th} element of the model mutation feature vector is set to 1; otherwise, it is set to 0. Through this method, we obtain the model mutation feature vector for t, represented as $(1, 0, \ldots, 1)$. This vector indicates that the first mutated model of M, denoted as m_1, predicts differently for the test t compared to the original model M. Conversely, the second mutated model of M, denoted as m_2, predicts the same for the test t as the original model M.
- **Feature Combination**: MLPrior concatenates the three types of features obtained from the previous steps (i.e., Original Attribute Features, Input Mutation Features, and Model Mutation Features) to form the final feature vector for t. This final feature vector is represented as (Original Attribute Features, Input Mutation Features, Model Mutation Features) = $(i_1, i_2, \ldots, i_n, 0, 1, \ldots, 0, 1, 0, \ldots, 1)$.

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

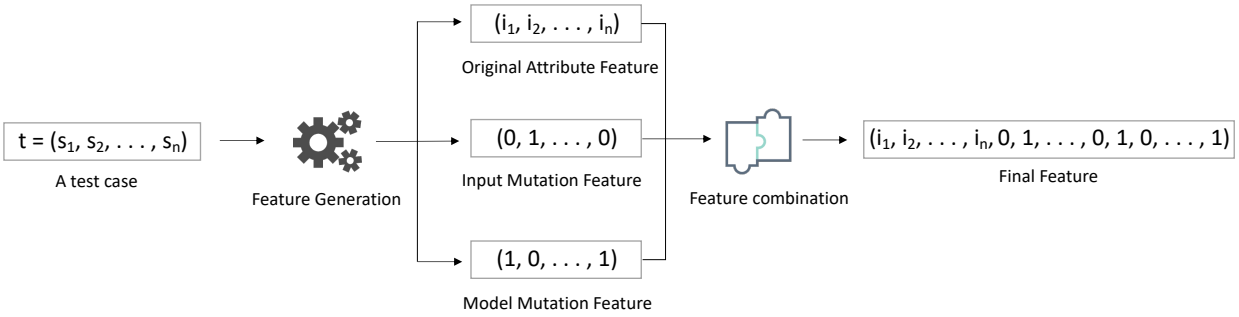

Fig. 2: Concrete example of feature generation in MLPrior

attribute value "male" is transformed into 1, while "female" is transformed into 0. The motivation behind extracting these original features is explained as follows.

Prior research [24] pointed out that *test inputs situated closer to the decision boundary of a model are more likely to be misclassified*. In order to effectively capture the spatial relationship between a test input and the decision boundary and to preserve the carefully-selected and low-dimensional features of the classical ML test set, we directly generate feature vectors of each input from its original attribute values.

3.2 Mutation Rule Specification

In this stage, we propose two types of mutation rules designed specifically for classical ML models and their corresponding datasets. The principle underlying our utilization of mutation testing in test prioritization is: *If a test input exhibits high sensitivity to the injected mutations, this input is more likely to detect faults in the system*. This principle is derived from previous research in traditional mutation testing [23], [61], [62]. We extend this principle to encompass ML systems, correspondingly designing model mutation rules and input mutation rules. The key insights of MLPrior are that: 1) if an input can kill many mutated models (i.e., the predictions for the input made by the mutated models and the original model are different), indicating that this input is sensitive to model mutations, MLPrior considers this input more likely to be misclassified. 2) If the prediction result for a given test input is different from that of many of its mutated inputs, indicating that the predictions for the input are sensitive to the mutations, MLPrior considers this input more likely to be misclassified. In the following sections, we provide a detailed explanation of our mutation approaches.

3.2.1 Model mutation rules

The model mutation rules are designed to make slight changes to the architecture parameters or weight parameters of the pre-trained ML models to generate mutated models. We ensure that the new parameter values are close to their original values in order to achieve slight mutations. It is important to note that this process does not involve any retraining operations. Therefore, the total execution time of generating model mutants is short, with an average duration of 3 seconds, as shown in Table 4.

In our study, we evaluated the effectiveness of MLPrior using five classical ML models, namely Decision Tree [9], K-Nearest Neighbors (KNN) [75], Logistic Regression (LR) [10], XGBoost [8], and Gaussian Naive Bayes (GaussianNB) [8]. The rationale behind selecting these models is twofold: 1) They have gained widespread adoption in various industries due to their interpretability and proven performance [32], [76]; 2) These models have been extensively utilized in recent ML testing studies [26]. It is important to note that MLPrior's applicability extends beyond the evaluated models. By making simple adjustments to the model mutation rules (i.e., enabling them to target the architecture parameters or weight parameters of the evaluated model), it can be adapted to a diverse range of interpretable ML models. We elaborate on the specific details of conducting model mutation as follows.

❶ **Decision Tree** [9] Decision tree is a machine learning method that predicts data step-by-step based on features. During prediction, attribute values are utilized to make decisions at internal nodes of the tree, determining which branch node to enter based on the decision outcome until a leaf node is reached to obtain the classification result.
Input to Decision Tree: The input to a Decision Tree consists of a dataset containing instances with associated features. The Decision Tree algorithm utilizes these input features to create a hierarchical structure that facilitates effective classification.
Process of Classification: Decision tree operates by sequentially making decisions at each split node of the tree. For a given input, it begins at the root node and evaluates the features of the input to determine the appropriate branch to follow at each split node. This process iterates until a leaf node is reached, signifying a classification outcome.
Mutating Decision Tree: To induce mutation in the Decision Tree model, we randomly select a set of split nodes and introduce random deviations to their threshold values, thereby influencing the predictive outcomes of the Decision Tree model. We explain below why changing the thresholds can alter the predictive results of a decision tree: Consider a situation where a given test sample t passes through nodes in the original tree. Based on decisions made at split nodes, it arrives at leaf node A, and thus will be classified as A category. After making slight adjustments to the thresholds of a group of decision nodes, when sample t traverses the mutated tree, the modified decision thresholds at split nodes can lead it to reach leaf node B.

❷ **K-Nearest Neighbors (KNN)** [75] KNN (K-Nearest

Neighbors) is a widely adopted classical machine learning model. It classifies an input into a class based on the majority class of its K nearest neighbors in the feature space.

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

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

❸ **Logistic Regression (LR)** [10] Logistic regression establishes a linear functional relationship to construct a connection between input features and probability outputs. It employs a Sigmoid function (as displayed in Formula 1) to map the results onto the [0, 1] interval, representing the probability of belonging to class 1. This enables the classification of input samples.

Weight Coefficient: In the Sigmoid function of logistic regression, weight coefficients determine the impact of different features on predicting the output. Each feature is assigned a corresponding weight coefficient. For example, in Formula 1, the weight coefficient for the feature x_0 is w_0.

Mutating Logistic Regression: To introduce mutation to the Logistic Regression, we randomly select a feature from the Sigmoid function and modify its weight coefficient, thus affecting the model's predictions. For example, consider Formula 1, which represents a trained Logistic Regression model taking four input features: x_0, x_1, x_2, and x_3. In this formula, w_0, w_1, w_2, and w_3 denote the weight coefficients for each feature, and $f(x)$ represents the prediction score. We mutate the model by randomly selecting one of the four weight coefficients and making a slight adjustment to its weight coefficient. This mutation process directly influences the output value of $f(x)$, consequently impacting the classification results of the model.

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

❹ **XGBoost** [8] XGBoost is a widely used gradient boosting algorithm designed for enhanced predictive modeling. XGBoost is a variant of the boosting algorithm [77], which aims to integrate multiple weak classifiers into a robust classifier. As a boosting tree model, XGBoost aggregates multiple tree models to form a powerful classifier. In binary classification tasks, XGBoost defaults to output 0 or 1, representing two different classes. Internally, XGBoost calculates an initial probability value p, subsequently comparing it to a threshold (with a default value of 0.5) prior to determining the final class output: values exceeding 0.5 yields an output of 1, whereas values below 0.5 yield an output of 0.

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

It can be observed that t_1 is more sensitive to the injected mutation than t_2, and we consider that t_1 is more likely to be misclassified by the model. This mutation rule can be reasonably interpreted from an uncertainty perspective: when a slight adjustment in the model's classification threshold can alter the test's classification result, it indicates that the model's prediction probability for that test is close to 0.5. According to prior work [16], the closer a prediction probability is to 0.5, the greater the model's uncertainty regarding that test, making it more prone to misclassification.

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

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

3.2.2 Input mutation rules

The prior work [24] introduced a mutation operator, *noise perturbation*, for mutating inputs in image format, which adds noise to data for mutation. A common type of image noise is occlusion noise [78], achieved by overlaying a black block on the part of the image. This black block typically consists of a matrix filled with 0. The method involves replacing the matrix of pixel values at the original location in the image with this zero-filled matrix (black block). Inspired by this technique, MLPrior's input mutation rule involves randomly selecting a specific feature from the feature vector of t and changing its value to 0. Before this, MLPrior initially converts all attributes of t into a corresponding numerical feature vector. The objective is to alter the attribute value of this particular feature, thus affecting the model's predictions. To gain a deeper insight into the impact of input mu-

tation rules on model predictions, we provide explanations using the five classical ML models evaluated in our study as examples. It is important to note that our input mutation rules are applicable to a wide range of datasets for classical ML models.

❶ **Decision Tree** Given a test input, if a specific feature value of this input is changed to 0, it could lead to a change in the decision path that the input takes down the tree. This mutation can cause the input to be categorized differently than it would have been without the mutation.

❷ **K-Nearest Neighbors (KNN)** For KNN, changing the value of a feature to 0 can alter the distance calculation between this input and other instances. This shift in distances can lead to a different set of k nearest neighbors being considered, thereby potentially affecting the classification result of the input.

❸ **Logistic Regression** In logistic regression, modifying a feature's value to 0 will impact the coefficients associated with that feature. This can lead to a different logistic function, causing the instance's predicted probability to shift, ultimately affecting the classification outcome.

❹ **XGBoost** For a given sample, setting a feature of it to 0 can influence the way that features contribute to the ensemble of decision trees. This can lead to different tree structures being emphasized during prediction, thereby affecting the final prediction of the sample.

❺ **Gaussian Naive Bayes (GaussianNB)** For a given sample, setting a feature's value to 0 can impact the calculation of probabilities for the various classes based on the Gaussian distribution assumption. This can influence the final classification result.

3.3 Mutation Feature generation

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

- **Input Mutation Features (IMF)** Based on the input mutation rules presented in Section 3.2.2, MLPrior generates a set of input mutants for each test $t \in T$. Subsequently, MLPrior proceeds to compare the predictions of model M for each input mutant with that of the original input t to construct the input mutation vector. During this process, if the prediction for the i-th mutated input differs from that of the original test input t, the corresponding i-th element of the feature vector is assigned a value of 1; otherwise, it is assigned a value of 0. An example of the resulting feature vector is (0, 1, ..., 0).
- **Model Mutation Features (MMF)** Based on the model mutation rules described in Section 3.2, MLPrior generates a set of mutated models for the original ML model M. For each test $t \in T$, MLPrior identifies whether t "kills" each of the mutated models (i.e., whether the predictions made by the mutated model and the original ML model for t are different) to construct the model mutation vector. More specifically, if t kills the i-th mutated model, the i-th element of t's model mutation vector will be set to 1. Otherwise, the i-th element will be set to 0. An example of the resulting feature vector is (1, 0, ..., 0).

3.4 Feature Concatenation

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

3.5 Learning-to-rank

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

Ranking model building MLPrior leverages the XGBoost ranking algorithm [8], an optimized distributed gradient boosting learning algorithm, to construct the ranking model. Given the classical ML model M with dataset D, we first split the dataset D into two partitions: the training set R and the test set T, in a 7:3 ratio [79]. The test set remains untouched for the purpose of evaluating MLPrior. Based on the training set R, our objective is to construct a training set R' for training the ranking models. To achieve this, we generate the final feature vector for each $r \in R$, following the steps described in Section **??** to Section 3.4. These features are used as the training features for the dataset R'. Next, we utilize the original ML model M to classify each instance $r \in R$ and then compare the model's predictions with the corresponding ground truth of r. By doing so, we can identify whether r is misclassified by the model M. If r is misclassified, we label it as 1; otherwise, we label it as 0. As a result, we obtain the labels for the training set R'. Based on the constructed training set and corresponding training labels obtained above, we can proceed to train the ranking model of MLPrior.

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

3.6 Variants of MLPrior

In order to explore the influence of different ranking models on the effectiveness of MLPrior, we propose four variants, denoted as MLPrior^T, MLPrior^K, MLPrior^L, and MLPrior^N. These variants utilize different ranking models for test prioritization, namely, decision tree [80], K-nearest neighbors (KNN)[29], logistic regression[10], and Gaussian Naive Bayes (GaussianNB) [81], respectively. They solely differ in the selection of the ranking models, while the remaining workflow is kept identical.

- **MLPrior^T** This variant incorporates the decision tree ranking model. The principle of the Decision Tree algorithm is to partition the dataset into subsets at split nodes, iteratively branching until reaching leaf nodes that provide the final classification.
- **MLPrior^K** integrates the KNN algorithm. KNN is a well-established machine learning technique. It operates on the fundamental principle of proximity, where the classification of a sample is determined by considering the majority labels of its K nearest neighbors in the feature space.
- **MLPrior^L** integrates the Logistic Regression algorithm [10]. Logistic Regression employs the logistic function to transform the linear combination of the independent variables into a range between 0 and 1. Consequently, this probability value is employed to perform classification.
- **MLPrior^N** integrates the Gaussian naive Bayes (GaussianNB) ranking model. GaussianNB is a probabilistic machine learning classification technique based on the Gaussian distribution. It assumes that each feature possesses independent predictive power for the output variable. The final prediction is obtained by combining the predictions derived from all features.

4 Study Design

4.1 Research Questions

Our experimental evaluation answers the research questions below.

- **RQ1: How does MLPrior perform in terms of effectiveness and efficiency?**
  To solve the labelling cost problem, we propose MLPrior, a test input prioritization approach specifically designed for classical ML models. In this research question, we evaluate the effectiveness and efficiency of MLPrior by comparing it with several existing test prioritization approaches [16], [20].
- **RQ2: How does MLPrior perform on different types of test inputs?**
  In order to evaluate the effectiveness of MLPrior in various scenarios, we constructed mixed noisy datasets and fairness datasets. We compare the effectiveness of MLPrior against various test prioritization approaches on the generated datasets.
- **RQ3: How do different ranking algorithms impact the effectiveness of MLPrior?**
  In MLPrior, we employ XGBoost [8] as the ranking model for test prioritization. In this research question, we investigate the impact of different ranking models on the effectiveness of MLPrior. To this end, we construct four variants employing different ranking models: decision tree [80], K-nearest neighbors (KNN)[29], logistic regression[10], and Gaussian Naive Bayes (GaussianNB) [81]. By evaluating the effectiveness of these variants, we explore the influence of ranking models.
- **RQ4: To what extent does each type of features contribute to the effectiveness of MLPrior?**
  To construct the feature vector for a given test input, MLPrior generates three types of features: model mutation features, dataset mutation features, and attribute features. In this research question, our objective is to investigate the extent to which each type of features contributes to the effectiveness of MLPrior.
- **RQ5: How does the selection of main parameters of MLPrior impact its effectiveness?**
  We investigate the influence of the main parameters in MLPrior. Our objective is to evaluate whether MLPrior can consistently outperform the compared test prioritization approaches when these main parameters fluctuate.

4.2 Subjects

In our research, we utilized 305 subjects to assess the effectiveness of MLPrior. A subject in this context refers to a combination of a classical ML model and a dataset. The description of these subjects can be found in Table 1. Out of the 305 subjects, 25 subjects (5 datasets $\times$ 5 ML models) were generated using natural datasets, while 250 subjects were generated using mixed noisy datasets. Additionally, 30 subjects were generated using fairness datasets. Below, we explain the construction method for mixed noisy datasets and fairness datasets.

- **Mixed noisy datasets** blend natural data with noisy data, with the natural data accounting fo r 70% and the noisy data accounting for 30%. The reason we chose 30% is that: A high noise ratio, such as 90%, would lead to a substantial proportion of noisy test inputs. In this scenario, a significant number of misclassified tests would be chosen by any prioritization method, making it difficult to demonstrate the effectiveness of MLPrior. Therefore, to ensure an effective evaluation of MLPrior and the compared approaches, we choose a reasonable noise generation ratio (i.e., 30%). For each of the five natural datasets, we generated 10 mixed noisy datasets, resulting in a total of 50 (5 $\times$ 10) mixed datasets. Each mixed dataset was paired with five classical ML models, leading to 250 subjects (50 datasets $\times$ 5 models).
- **Fairness datasets** refer to datasets carefully constructed with a specific focus on avoiding the introduction of biases related to individual attributes, such as gender, age, etc. In our study, we generated a fairness dataset from a natural dataset following the approach utilized in prior work [26]: we randomly selected a subset of instances and modified their gender and age attribute values while keeping their original labels untouched. Employing this approach, we generated 6 fairness datasets. We pair each dataset with five classical ML models, leading to 30 subjects (6 datasets $\times$ 5 models).

4.2.1 Datasets

TABLE 1: Classical ML models and datasets

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

In our study, we evaluate MLPrior using five datasets: Adult [82], Bank [83], Stroke [84], Diabetes [85] and Heartbeat [86]. The reason for selecting these five datasets lies in their widespread utilization in the field of machine learning. Moreover, these datasets have been extensively employed in multiple recent research on classical machine learning testing, including 2022 FSE [26] and 2022 ICSE [87], [88], [89].

- **Adult** [82], [90], [91]: The adult dataset is designed to predict whether an individual's annual income exceeds 50K based on various demographic and financial attributes. It consists of 48,842 instances, with each instance representing a single individual. All the instances are divided into two classes: >50K and <=50K. Each individual is described by 14 different attributes, such as age, occupation, education level, workclass, etc.
- **Bank** [83], [90]: The bank dataset is utilized to forecast whether a client will subscribe to a term deposit, utilizing their demographic, financial, and social information. It consists of 49,732 instances, classified into two classes: subscribing to the term deposit or not subscribing. Each instance encompasses 16 attributes, such as age, education, loan, and balance.
- **Stroke** [84]: The stroke dataset is employed for predicting the occurrence of a stroke in patients. It comprises 40,907 instances, classified into two classes: having a stroke or not having a stroke. Each instance is described using 10 attributes, such as age, heart disease, hypertension, work type, residence type, and smoking status.
- **Diabetes** [85]: The diabetes dataset is utilized for predicting diabetes occurrence in patients. It comprises 253,680 survey responses related to diabetes. This dataset is categorized into three classes: 0 for no diabetes or diabetes only during pregnancy, 1 for prediabetes, and 2 for diabetes.
- **Heartbeat** [86]: The Heartbeat dataset is used for classifying heartbeat signals. In our experiments, we used 30,000 heartbeat signal sequence data. Each sample in the dataset has a consistent sampling frequency and equal length in its signal sequence. The Heartbeat dataset is divided into 4 classes, which are categorized as heartbeat signal types (0, 1, 2, 3).

4.2.2 Classical ML models

We evaluate the effectiveness of MLPrior using five well-established classical ML models: Decision Tree [9], K-Nearest Neighbors (KNN) [75], Logistic Regression (LR) [10], XGBoost [8], and Gaussian Naive Bayes (GaussianNB) [8]. These models were chosen based on two primary reasons: First, their widespread adoption in various industries owing to their interpretability and demonstrated performance [12], [32], [76], [92].

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

- **Hospitality industry** [76] The **logistic regression** model can utilize financial data to predict whether a hotel business is at risk of bankruptcy. Investors in the hotel industry will rely on these models to make crucial financial and operational decisions. If the predictions are inaccurate, Investors can make erroneous investment decisions, such as investing in businesses that are at risk of bankruptcy.
- **Service industry** [32] The **decision tree** model can be employed to analyze the impact of information and communication technology (ICT) on service industry performance using global service industry data from the World Bank. Service industry companies will depend on such analyses to formulate strategies, such as investing in ICTs. Incorrect predictions could result in misallocation of resources, affecting the company's long-term performance and competitiveness.
- **Financial industry** [93], [94] The **XGBoost** algorithm can be utilized for personal credit risk assessment. Rao *et al.* [93] employed XGBoost to predict an individual's credit risk for determining loan approval decisions. Moreover, **KNN** can be used for credit scoring(i.e., assessing the credit risk of loan applications) [94].
- **Healthcare industry** [95] The **Gaussian Naive Bayes** model can be leveraged for diagnosing cancer based on the patient's medical information [95].

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

Second, their extensive use in recent ML testing studies [26], [96], [97], [98], [99]. Importantly, it should be noted

that MLPrior's applicability is not limited to the evaluated models. With minor adjustments to the model mutation rules (i.e., making them target the architecture parameters or weight parameters of the assessed ML model), MLPrior can be adapted to various interpretable ML models.

- **XGBoost** [8] XGBoost, an ensemble method that belongs to the family of boosting algorithms, functions by integrating the forecasts of multiple Classification and Regression Trees (CART) [100] to create a robust classification mechanism. This algorithm amalgamates weak learners to engineer a powerful model with superior predictive capacity.
- **Gaussian Naive Bayes (GaussianNB)** [95] Gaussian Naive Bayes, a probabilistic classifier based on Bayes' theorem with an assumption of independence among predictors, is known for its efficacy in multiclass classification problems and its robustness against irrelevant features.
- **Logistic Regression (LR)** [10] Logistic Regression is a widely-adopted statistical model employed in scenarios of binary classification tasks. This model is founded on the principles of probability and logistic function, offering an interpretable mathematical framework.
- **Decision Tree** [9] Decision tree constructs a tree-like structure, where internal nodes represent decision points based on feature values, and leaves represent the predicted outcomes.
- **K-nearest neighbors (KNN)** [75] KNN is a widely-adopted classification algorithm that assigns labels to instances based on the majority vote of their K neighboring data points. The KNN algorithm is known for its simplicity and flexibility in handling classification tasks.

4.3 Compared Approaches

To demonstrate the effectiveness of MLPrior, we compared it with multiple test prioritization approaches. The considered methods include DeepGini (ISSTA 2020) [16], VanillaSM (ISSTA 2022) [20], Prediction-Confidence Score (ISSTA 2022) [20], and Entropy (ISSTA 2022) [20]. We select these comparative methods because 1) they can be adapted to classical ML models for test prioritization; 2) their effectiveness on DNNs has been demonstrated [20], [16].

- **DeepGini** [16] DeepGini operates by assessing the model's uncertainty in its predictions for tests. The fundamental premise of DeepGini is that tests for which the model exhibits greater uncertainty in its predictions are deemed to have a higher likelihood of being incorrectly predicted. Consequently, these tests will be prioritized higher. The mechanism for calculating this uncertainty in DeepGini is encapsulated in a specific formula, referred to as Formula 2. In this formula, the symbol $\xi(t)$ denotes the model's uncertainty regarding its prediction for a particular test t. The higher the value of $\xi(t)$, the greater the uncertainty associated with the model's prediction for the test t, and t will be prioritized higher. By prioritizing tests with higher values of $\xi(t)$, DeepGini can identify and prioritize test inputs that are potentially misclassified.

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

where N is the number of classes, and $p_{t,i}$ denotes the probability of the model predicting t belonging to class i.

- **VanillaSM** [20] The VanillaSM algorithm ranks all the tests by computing the difference between the highest activation probability within the output softmax layer for each test and 1. The calculation is defined by Formula 3. A lower value of $V(t)$ indicates that the test is more likely to be misclassified by the model.

$$V(t) = 1 - \max_{i=1}^{N} l_i(t) \tag{3}$$

where N is the number of classes. $\max_{i=1}^{N} l_i(t)$ represents the model's prediction probability for the most confident classification of test t among all N classes.

- **Prediction-Confidence Score (PCS)** PCS [20] prioritizes test inputs by calculating the difference between the probabilities of the model's most confident class and the second most confident class for each test. The formula is given as Formula 4. A smaller PCS(t) indicates that a test is more likely to be mispredicted by the model.

$$PCS(t) = p^1(t) - p^2(t) \tag{4}$$

where $p^1(t)$ is the predicted probability of the model for the most confident class of test t, and $p^2(t)$ is the predicted probability of the model for the second most confident class of test t.

- **Entropy** Entropy [20] ranks all tests by calculating the entropy value of the model's predicted probability vector for each test. A higher entropy value for a test indicates that it is more likely to be mispredicted by the model.
- **Random selection** [101] In random selection, the order of test input execution is determined randomly.

4.4 Measurements

Following the existing work [16], we employed two metrics to evaluate the effectiveness of MLPrior, the compared approaches, and the variants of MLPrior: Average Percentage of Fault Detection (APFD) [57] and Percentage of Faults Detected (PFD) [16].

- **Average Percentage of Fault-Detection (APFD)** APFD is a well-established metric utilized for evaluating the effectiveness of test prioritization. A higher APFD value indicates greater effectiveness. The APFD values are computed using Formula 5.

$$APFD = 1 - \frac{\sum_{i=1}^{k} o_i}{kn} + \frac{1}{2n} \tag{5}$$

where n denotes the number of test inputs in the test set, and k represents the number of misclassified inputs. o_i is the index of the i_{th} misclassified test within the prioritized test set. Below, we explain from a formula perspective why larger APFD values indicate high test prioritization effectiveness.

Firstly, in the formula, since n is a constant, a larger APFD value means that the value of $\sum_{i=1}^{k} o_i$ (i.e., the total index sum of misclassified tests within the prioritized list) is smaller. A smaller $\sum_{i=1}^{k} o_i$ implies that the misclassified tests are relatively positioned toward the front of the prioritized test set. This indicates that the misclassified tests are indeed prioritized at the beginning of the test set through the test prioritization approach, thus demonstrating that its effectiveness is high. Follow-

ing prior work [16], we normalize the APFD values to [0,1]. A prioritization approach is considered better when the APFD value is closer to 1.

- **Percentage of Fault Detected (PFD)** PFD quantifies the ratio of detected misclassified test inputs to the total number of misclassified tests. A higher PFD value suggests that a test prioritization approach is more effective. The calculation of PFD follows Formula 6.

$$PFD = \frac{\#F^d}{\#F} \tag{6}$$

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

4.5 Implementation and Configuration

In terms of the compared approaches, we employed the available implementations provided by their respective authors [20], [16]. Concerning the XGBoost ranking model, we utilized XGBoost version 1.4.2 [8]. For the ranking models Decision Tree, KNN, Logistic Regression, and GaussianNB, we utilized the package provided by scikit-learn 0.24.2 [102]. Regarding the parameters of the ranking models, we set the $n\_estimators$ parameter of XGBoost to 100. We set the max_iter parameter of Logistic Regression to 100. For the Decision Tree ranking algorithm, we set the $min\_samples\_split$ parameter to 2. The $var\_smoothing$ parameter of GaussianNB was set to 1e-9. Additionally, we set the $n\_neighbors$ parameter of KNN to 5.

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

5 Study Results

5.1 RQ1: Effectiveness and Efficiency of MLPrior

Objectives: We evaluate the effectiveness and efficiency of MLPrior in prioritizing test inputs for classical ML models.
Experimental design: We conducted experiments to evaluate the performance of MLPrior from three perspectives:

- **Effectiveness** To assess the effectiveness of MLPrior, we carefully designed 15 subjects consisting of three prevalent datasets, each paired with five classical ML models. Detailed information regarding the subjects can be found in Table 1. Moreover, we compared MLPrior against a range of DNN prioritization approaches, namely DeepGini [16], Vanilla Softmax [20], Prediction-Confidence Score (PCS) [20], Entropy [20], and Random Selection. To measure the effectiveness, we used the APFD metric [57] and the PFD metric [16], which are widely-adopted measures for evaluating test prioritization techniques.
- **Efficiency** We evaluate the efficiency of MLPrior by quantifying the time required for each step of MLPrior, as well as the time cost of each compared approach.
- **Statistical analysis** Considering the randomness associated with the training process of the ML models and the MLPrior approach, we conduct a statistical analysis to ensure the stability of our research. More specifically, we replicated all the experiments a total of five times, calculating average results to report in this section. Furthermore, we calculated the p-values to evaluate the statistical significance of our findings.

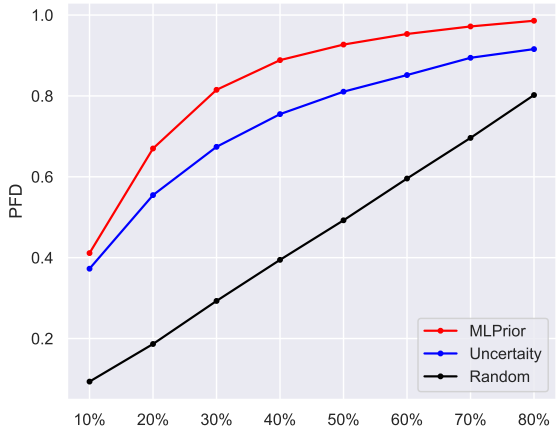

Fig. 3: Test prioritization effectiveness among MLPrior and the compared approaches (dataset Bank with model GaussianNB). X-Axis: the percentage of prioritized tests; Y-Axis: the percentage of detected miscalssified tests

Results: The experimental results pertaining to RQ1 are presented in Table 2, Table 3, Figure 3, Table 4 and Table 5. We highlight the approach with the highest effectiveness in grey to facilitate quick and easy interpretation of the results.

When applied to natural inputs, MLPrior outperforms all the compared methods in terms of APFD across all subjects, with an average improvement of 14.74%∼66.93% over the compared approaches. Table 2 exhibits the effectiveness of MLPrior in comparison to the compared test prioritization approaches across different subjects. From the table, we see that MLPrior outperforms all the compared methods across all subjects. Specifically, the APFD values of MLPrior range from 0.787 to 0.990, while that of the compared approaches span from 0.494 to 0.837. Table 3 demonstrates the effectiveness of MLPrior and the comparative test prioritization methods on multiclass classification datasets. We see that in all cases, the effectiveness of MLPrior is higher than all the comparative methods. Specifically, the APFD range of MLPrior is from 0.639 to 0.915, while the APFD range for the comparative methods is from 0.475 to 0.852. The experimental results demonstrate that MLPrior's effectiveness surpasses all comparative methods on multi-class datasets.

Table 5 shows the comparison of effectiveness between MLPrior and other test prioritization methods on all subjects in both binary and multi-class datasets. The evaluation

TABLE 2: Effectiveness comparison among MLPrior and DNN test prioritization approaches in terms of APFD on natural datasets (Binary Classification)

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

TABLE 3: Effectiveness comparison among MLPrior and DNN test prioritization approaches in terms of APFD on natural datasets (Multiclass classification)

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

TABLE 4: Time cost of MLPrior and the compared test prioritization approaches

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

TABLE 5: Effectiveness improvement of MLPrior over the compared approaches in terms APFD on natural datasets

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

metrics include the number of cases where each method performs the best (denoted as **#Best cases**), the average APFD value of each test prioritization approach (denoted as **Average APFD**), and the improvement of MLPrior relative to each comparison method (denoted as **Improvement(%)**). From Table 5, we can see that MLPrior performs the best across all cases, whether in binary or multi-class datasets. In binary datasets, the average APFD of MLPrior is 0.854. In multi-class datasets, it is 0.812, and across all subjects (including both binary and multi-class), it is 0.833. The average APFD of comparison methods across all subjects ranges from 0.499 to 0.726.

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

Figure 3 provides a visual comparison between MLPrior and other test prioritization approaches in terms of PFD on the Bank dataset with the GaussianNB model. In this figure, the effectiveness of MLPrior is represented by the red curve, while the blue curve represents the effectiveness of confidence-based test prioritization methods. Additionally, the black curve depicts the baseline effectiveness. It is noteworthy to mention that all confidence-based approaches are consolidated into a single line due to their identical effectiveness across all cases, as evidenced in Table 2.

The reason why all confidence-based methods yield the same experimental results on binary classification ML models is as follows: Given a binary classification model, suppose the probability of a test t belonging to category 1 is p, then the probability of it belonging to the other category is $1-p$. Regardless of the confidence-based method used, tests with p values close to 0.5 are deemed more uncertain [16] and thus are prioritized to the front. Therefore, the experimental results of all test prioritization methods are the same. We explain this in detail below.

Feng *et al.* [16] demonstrated that in a binary classification model, if the model's prediction probability for a test is $(0.5, 0.5)$, it means the model is most uncertain about this test, indicating this test is more likely to be misclassified. The closer a test's p value is to 0.5, the more uncertain the model is about that particular test. Consequently, uncertainty is solely determined by p. Regardless of the specific confidence-based test prioritization method employed, tests with p values closer to 0.5 will be prioritized over others.

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

From Figure 3, we see that MLPrior consistently outperforms all the compared methods across different prioritization ratios. These experimental results strongly suggest that MLPrior exhibits higher effectiveness than other test prioritization approaches in classical ML test prioritization. As stated in the experimental design, due to the inherent randomness associated with the training process, we conducted a statistical analysis. This analysis involved repeating all experiments a total of five times. The p-value of the experimental results was found to be significantly less than 10^{-06}, which suggests that MLPrior can stably outperform the compared test prioritization approaches.

MLPrior showcases acceptable efficiency, with an average execution time of less than 20 seconds. In addition to evaluating its effectiveness, we also compared the efficiency of MLPrior with other test prioritization approaches, and the experimental results are presented in Table 4. The findings indicate that the average total running time of MLPrior on each subject is under 20 seconds, which can be broken down into three main components: feature generation (3 seconds), ranking model training (15 seconds), and prediction (55.133 ms). Here, 'ms' refers to milliseconds. The prediction times for the confidence-based test prioritization methods are as follows: DeepGini: 1.323 ms; VanillasM: 1.020 ms; PCS: 1.355 ms; Entropy: 114.483 ms. While confidence-based test prioritization techniques exhibit higher efficiency with a running time of less than 1 second, the computational cost of MLPrior remains reasonable in practical scenarios, especially considering the laborious and costly nature of manual labeling. Despite being slightly less efficient than confidence-based methods, the considerable improvement in effectiveness demonstrated by MLPrior, ranging from 18.78% to 70.46% compared to those techniques, underscores its overall performance.

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

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

Objectives: In addition to assessing MLPrior's performance on natural test sets, we also evaluate its effectiveness on different types of test inputs, encompassing mixed noisy data and fairness data. *Mixed noisy datasets* are composed of 70% natural data and 30% of noisy data. *Fairness datasets* are constructed with the aim of avoiding biases associated with individual attributes, such as gender and age. Ensuring fairness in machine learning is crucial to prevent bias and discrimination against specific groups during predictions. Fairness has emerged as a critical ethical consideration across diverse machine learning domains, such as recruitment, loan approvals, and medical diagnosis [25]. In these domains, the absence of fairness can result in unjust treatment of certain groups, significantly impacting individuals' lives and rights.

Our investigation revolves around two primary sub-questions:

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

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

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

[Experiment ❷] To generate fairness data for evaluation, we adopt the approach used in previous research [26]. Specifically, for each natural dataset utilized in RQ1 (i.e., Adult, Bank, and Stroke), we randomly selected a subset of instances from the original test set and modified their gender and age attribute values while keeping the original labels untouched. The reason for ensuring the labels untouched is as follows: In the context of ensuring fairness, the model should maintain consistent classification results when the protected attributes (such as genders and ages) are changed, while all other attributes remain unaltered.

Concretely, for the attribute "gender", we changed half of the "male" to "females" and half of the "females" to "males". Regarding the attribute "age", following the prior work [87], we modified the "middle age" (30∼59) instances in the test set to "young age" (18∼29) while converting the "young age" test instances to "middle age." Using the

TABLE 6: Effectiveness comparison among MLPrior and DNN test prioritizatiion approaches in terms of APFD on mixed noisy datasets (Binary Classification)

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

TABLE 7: Effectiveness comparison among MLPrior and DNN test prioritizatiion approaches in terms of APFD on mixed noisy datasets (Multiclass classification)

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

TABLE 8: Effectiveness improvement of MLPrior over the compared approaches in terms of APFD on mixed noisy datasets

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

generated fairness test sets, we compare the effectiveness of MLPrior with other test prioritization methods.

Results: The experimental findings pertaining to RQ2.1 are presented in Table 6, Table 7, Table 8. Table 6 showcases the effectiveness difference between MLPrior and the compared test prioritization methods when applied to mixed noisy inputs. The evaluation metric employed is the Average Percentage of Faults Detected (APFD). We highlight the approach with the highest effectiveness in grey to facilitate easy interpretation of the results.

On mixed noisy inputs, MLPrior consistently performs better than all the compared approaches, with an average improvement of 18.55%∼67.73%. From Table 6, we see that MLPrior consistently outperforms all the compared methods across each case. Remarkably, the APFD values achieved by MLPrior range from 0.810 to 0.982, while that of the compared methods range from 0.497 to 0.766. Table 7 presents the effectiveness of MLPrior compared to other test prioritization methods on noisy datasets for multiclassification. We see that MLPrior outperforms all other test prioritization methods across all multiclassification subjects. The range of APFD values for MLPrior is from 0.639 to 0.916, whereas the range for the compared test prioritization methods is from 0.485 to 0.851. We conclude that on noisy datasets for multiclassification, the effectiveness of MLPrior surpasses that of the compared test prioritization methods.

Table 8 provides an overall comparison of the effectiveness of MLPrior and other test prioritization methods on binary classification datasets, multiclass classification datasets, and all subjects (both binary and multiclass). The evaluation metrics include the number of cases where each method performs the best (denoted as **#Best cases**), the average APFD value of each test prioritization approach (denoted as **Average APFD**), and the improvement of MLPrior relative to each comparison method (denoted as **Improvement(%)**).

In Table 8, we observe that MLPrior performs the best across all subjects, regardless of whether they are binary or multiclass. The average APFD of MLPrior on all subjects (including both binary and multiclass) is 0.837. Specifically, the average APFD of MLPrior in binary classification is 0.858, while in multiclass classification, it is 0.815. In contrast, the range of the average APFD for the comparison methods across all subjects is from 0.499 to 0.706. Moreover, across all subjects, the average improvement of MLPrior relative to the comparison test prioritization methods ranges from 18.55% to 67.73%.

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

The experimental results of RQ2.2 are presented in Table 9, Table 10, Table 11. Table 9 displays the effectiveness differences between MLPrior and all the comparative meth-

TABLE 9: Effectiveness comparison among MLPrior and DNN test prioritization approaches in terms of APFD on fairness datasets (Binary Classification)

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

TABLE 10: Effectiveness comparison among MLPrior and DNN test prioritization approaches in terms of APFD on fairness datasets (Multiclass classification)

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

TABLE 11: Effectiveness improvement of MLPrior over the compared approaches in terms of APFD on fairness datasets

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

ods on the fairness dataset in terms of APFD. The gray shading indicates the best-performing method for each case.

On fairness data, MLPrior consistently performs better than all the compared approaches, with an average improvement of 15.34%∼62.72%. We see that MLPrior achieves the highest effectiveness across all cases, with an APFD range of 0.813 to 0.897. In contrast, the comparative methods have an APFD range of 0.484 to 0.788.

Table 10 showcases the effectiveness of MLPrior compared to other test prioritization methods on fairness datasets for multiclassification. We can see that MLPrior exceeds the performance of all other test prioritization methods in all multiclassification subjects. The APFD values for MLPrior range from 0.765 to 0.801, while the compared test prioritization methods range between 0.495 and 0.759. The experimental results demonstrate that, in the context of fairness datasets for multiclassification, MLPrior's effectiveness is superior to that of the other compared test prioritization methods.

Table 11 presents a comparative analysis of the effectiveness between MLPrior and other test prioritization methods across all fairness subjects within binary and multi-class datasets. The evaluation metrics encompass the number of instances where each method is most effective (denoted as **#Best cases**), the average APFD value for each test prioritization approach (denoted as **Average APFD**), and the relative improvement of MLPrior compared to each method (denoted as **Improvement(%)**). According to Table 11, MLPrior consistently outperforms other methods in all scenarios, whether in binary or multi-class datasets. Specifically, in binary datasets, MLPrior's average APFD is 0.847. In multi-class datasets, it is 0.776, and the overall average across all subjects (encompassing both binary and multi-class) stands at 0.812. The average APFD for the comparison methods across all subjects varies from 0.499 to 0.704.

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

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

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

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

Experimental design: In order to investigate the impact of different ranking models, we propose four variants of MLPrior denoted as MLPrior^T, MLPrior^K, MLPrior^L,

and MLPrior^N. These variants employ the ranking models decision tree [80], K-nearest neighbors (KNN)[29], logistic regression[10], and Gaussian Naive Bayes (GaussianNB) [81], respectively. The only difference between them and the original MLPrior lies in the selection of the ranking models, while the rest of the workflow remains unchanged. We utilize the APFD metric to evaluate the effectiveness differences of MLPrior, these variants, and the comparative test prioritization methods on natural and mixed noisy datasets.

Results: The experimental results for RQ3 are presented in Table 12, Table 13, Table 14, Table 15, Table 16 and Table 17. Tables 12 and Table 13 display the effectiveness of MLPrior, its variants, and the compared test prioritization methods on natural datasets. Tables 14 and Table 15 show their effectiveness on noisy datasets. Table 16 presents their effectiveness on fairness datasets. Table 17 illustrates their average performance across all datasets (including natural, noisy, and fairness datasets), as well as the improvements of MLPrior relative to its variants and the compared test prioritization methods.

MLPrior outperforms all its variants in test prioritization, indicating that among all ranking models, the XGBoost model (utilized by the original MLPrior) can better utilize the generated features of test inputs for test prioritization. Table 12 and Table 13 demonstrate the effectiveness of MLPrior on **natural** datasets, including binary classification datasets (Table 12) and multiclass classification datasets (Table 13). We see that, whether on binary or multiclass datasets, the effectiveness of MLPrior (measured by APFD) consistently surpasses all its variants. On binary natural datasets (Table 12), the APFD range for MLPrior is from 0.8110 to 0.990, while the range for its variants is from 0.589 to 0.898. On multiclass natural datasets (Table 13), the APFD range for MLPrior is from 0.639 to 0.915, while the range for its variants is from 0.580 to 0.890. We conclude that on **natural** datasets, the effectiveness of MLPrior exceeds all its variants. Moreover, on **noisy** datasets, encompassing both binary classification datasets (Table 14) and multiclass classification datasets (Table 15), MLPrior also consistently outperforms all its variants across all cases.

Table 16 demonstrates the effectiveness of MLPrior, its variants, and the compared test prioritization methods on **fairness** datasets. We see that, whether on fairness datasets constructed based on age or those constructed based on gender, the effectiveness of MLPrior outperforms both its variants and all test prioritization methods. Specifically, the APFD range for MLPrior is from 0.765 to 0.897, while the range for its variants is from 0.648 to 0.821. We conclude that, on **fairness** datasets, the effectiveness of MLPrior exceeds all its variants.

Table 17 displays the effectiveness of MLPrior, its variants, and the compared test prioritization methods across all datasets (i.e., natural, noisy, and fairness datasets). We see that MLPrior performs the best across all 305 cases. Specifically, these 305 cases represent 25 natural subjects + 250 noisy data subjects + 30 fairness data subjects, totaling 305 cases. The detailed origins of these numbers can be referred to in Section 4.2. Additionally, across all subjects, the average effectiveness of MLPrior is 0.827, while the average effectiveness of its variants ranges from 0.683 to 0.770. Furthermore, the improvement of MLPrior over its variants in terms of APFD lies between 7.40% and 21.08%.

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

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

5.4 RQ4: Feature contribution Analysis

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

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

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

Results: Table 18 presents the contributions of different feature types to the effectiveness of MLPrior. In this table, we utilize the abbreviations MMF, IMF, and OAF to represent model mutation features, input mutation features, and original attribute features, respectively. The numbers after the feature abbreviations denote the indices of the corresponding features. For instance, *IMF-123* denotes the input mutation feature with index 123. We conducted the feature contribution analysis on both binary classification datasets (Adult, Bank, and Stroke) and multiclass classification datasets (Diabetes and Heartbeat).

All three types of features (i.e., model mutation features, input mutation features, and original attribute features) visibly contribute to the effectiveness of MLPrior. In Table 18, we find that in binary classification datasets, for the majority of cases (14 out of 15), all three types of features are present among the top-N most contributing features.

TABLE 12: Effectiveness comparison among MLPrior, MLPrior Variants and DNN test prioritization approaches in terms of APFD on natural datasets (Binary Classification)

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

TABLE 13: Effectiveness comparison among MLPrior, MLPrior Variants and DNN test prioritization approaches in terms of APFD on natural datasets (Multiclass classification)

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

TABLE 14: Effectiveness comparison among MLPrior, MLPrior Variants, and DNN test prioritization approaches in terms of APFD on mixed noisy datasets (Binary Classification)

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

TABLE 15: Effectiveness comparison among MLPrior, MLPrior Variants and DNN test prioritization approaches in terms of APFD on mixed noisy datasets (Multiclass classification)

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

TABLE 16: Effectiveness comparison among MLPrior, MLPrior Variants and DNN test prioritization approaches in terms of APFD on fairness datasets (Binary Classification & Multiclass classification)

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

TABLE 17: Effectiveness improvement of MLPrior over MLPrior Variants, and DNN test prioritization approaches

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

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

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

5.5 RQ5: Impact of Main Parameters in MLPrior

Objectives: We delve into the impact of main parameters on the effectiveness of MLPrior.

Experimental design: Building upon the existing research by Wang *et al.* [15], We delve into an exploration of the impact of three main parameters within the MLPrior's ranking model. These parameters include max_depth, which denotes the maximum tree depth for each XGBoost model, $colsample\_bytree$, representing the sampling ratio of feature columns during the tree construction process, and $learning\_rate$, indicating the boosting learning rate utilized in the XGBoost ranking model. To achieve the research objectives, we conducted a series of experiments using natural datasets. We carefully modified the aforementioned three main parameters and observed the variations in the effectiveness of MLPrior (measured by APFD).

Results: The experimental results of RQ5 are presented in Figure 4, illustrating the fluctuations in MLPrior's effectiveness when the main parameters' values are altered. The X-axis represents the parameter values, while the Y-axis represents MLPrior's effectiveness (measured by APFD). The solid red line corresponds to MLPrior, while the dashed lines represent the confidence-based test prioritization approaches. We investigated the influence of the main parameter on MLPrior's effectiveness across both binary classification datasets (Adult, Bank, and Stroke) and multiclass classification datasets (Diabetes and Heartbeat).

MLPrior consistently outperforms the confidence-based test prioritization approaches, even when the values of the main parameters are altered. Notably, we see that MLPrior consistently outperforms the confidence-based test prioritization methods across all subjects, as evidenced by the red line persistently positioned above the blue dashed lines. For example, in Figure 4(e), we observe that when the parameter $colsample\_bytree$ varies, MLPrior's APFD ranges from 0.86 to 0.88, whereas the confidence-based methods' APFD effectiveness is approximately 0.75. Moreover, under the multiclass dataset Heartbeat, when the parameter colsample_bytree changes, MLPrior's APFD ranges from around 0.84 to 0.85, whereas the APFD effectiveness of confidence-based methods ranges from around 0.745 to 0.750. Under the multiclass dataset Diabetes, when the

TABLE 18: Top-10 most contributing features for each subject

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

parameter learning_rate changes, MLPrior's APFD ranges from around 0.77 to 0.78, whereas the APFD effectiveness of confidence-based methods is around 0.71.

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

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

6 Discussion

6.1 Generality of MLPrior

While we employed five ML models in our study, MLPrior can actually be adapted for a broad range of classical ML models through simple modifications to the model mutation rules, specifically by enabling them to target the architecture parameters or weight parameters of the evaluated model. We explain below why MLPrior exhibits

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3350019

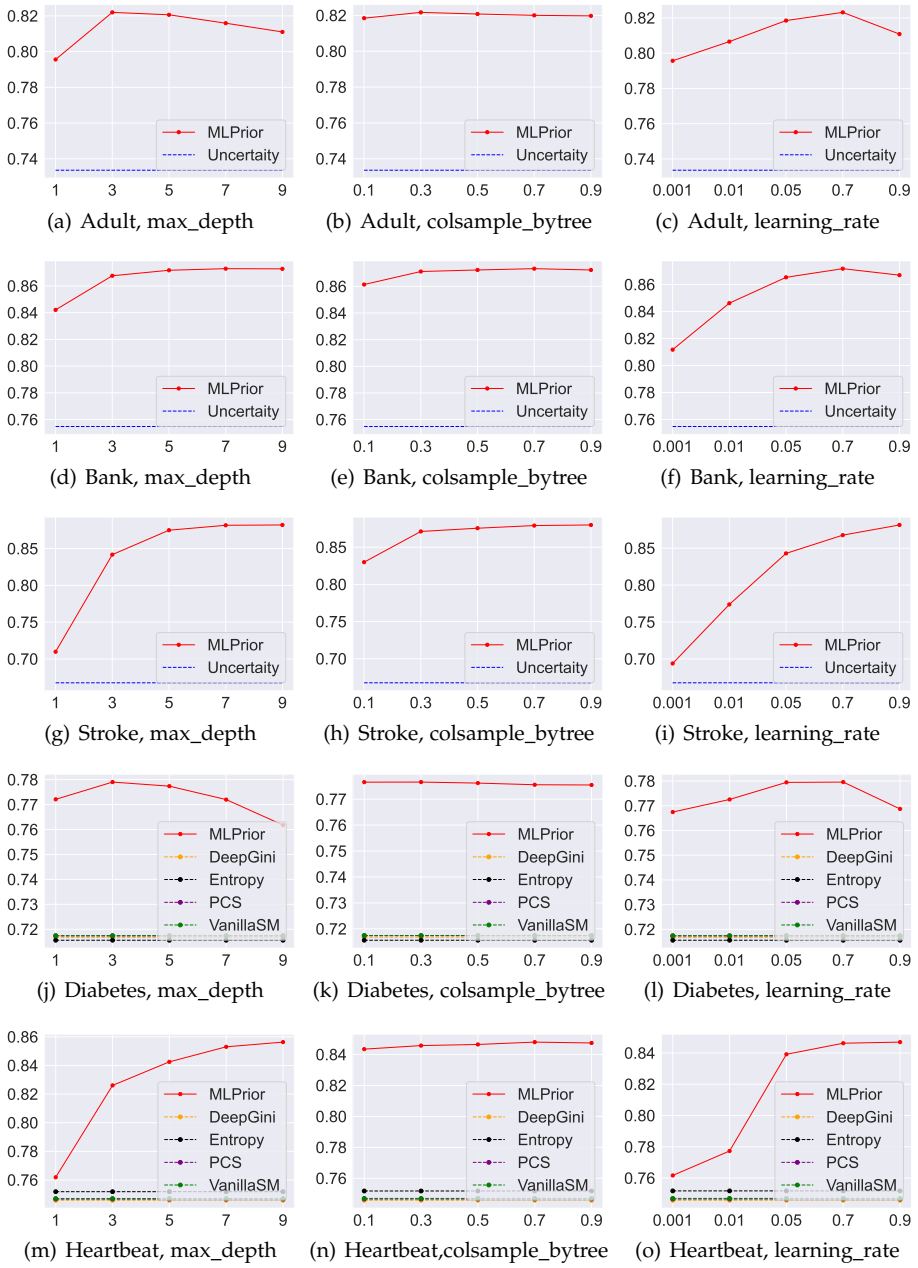

Fig. 4: Impact of main parameters in MLPrior

generality. First, the core element of MLPrior is feature generation, which involves generating three essential types of features from the target tests: Model mutation features, Input mutation features, and Attribute features. Once the features are generated, MLPrior can utilize the ranking model to learn from these features for the purpose of test prioritization. Concerning model mutation features, making the aforementioned simple adjustments (i.e., enabling model mutation rules to target the architecture parameters or weight parameters of the evaluated model) can allow for the generation of model mutation features. For input mutation features and attribute features, MLPrior is capable of directly generating these features. Consequently, MLPrior can be applied to a diverse range of classical ML models.

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

- **Skills needed to apply MLPrior to new ML models** When an ML testing practitioner aims to apply MLPrior to a new type of ML model, they need to possess the following skills: 1) An understanding of the internal parameters and mechanisms of the new machine learning model, to effectively carry out model mutation operations in accordance with MLPrior's methodology; 2) Basic Python knowledge to replace the functions for model mutation of the new type model with those for the original model; 3) Since input mutation and attribute feature generation are already designed as automatic pipelines, the testing practitioner can directly execute them without needing additional skills.
- **Characteristics for models to utilize MLPrior** When a model exhibits the following characteristics, it can be

added to the set of models that can use MLPrior: 1) The dataset of the model is in the tabular format, as our input mutation and attribute feature generation operations are specifically crafted for classical ML models that utilize tabular datasets; 2) The model is a white-box model, which allows for modifications to its internal structure or parameters, facilitating the implementation of MLPrior's model mutation operations.

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

- **Model Mutation Feature (MMF) Generation** To generate the model mutation feature for a test input, the following process should be executed: **1) Parameter Selection** Following the methodology of MLPrior's model mutation rules (i.e., modifying the architectural parameters or weight parameters of the model), the ML testing practitioner needs to select appropriate model parameters for the purpose of mutation and replace the previous model mutation function with the new model mutation function. **2) Automatic Pipeline** Once the replacement is complete, all other parts are automated pipelines, and MLPrior can automatically generate model mutation features.
- **Original Attribute Feature (OAF) Generation** The ML testing practitioner can directly obtain the OAF by implementing MLPrior, as we have designed an automated pipeline for the original feature transformation. This pipeline is capable of supporting new datasets in tabular format.
- **Input Mutation Feature (IMF) Generation** The ML testing practitioner can directly acquire the IMF by implementing MLPrior. MLPrior can automatically perform input mutations to obtain mutation features. This pipeline is capable of supporting new datasets in tabular format.

6.2 Threats to Validity

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

Threats to External Validity. The external threats to validity arise from the ML models and test datasets employed in our study. To mitigate these threats, we carefully selected a variety of ML models and datasets that are utilized by several top-level conferences [26], [89], [103] in the field of ML testing. Moreover, our evaluation of MLPrior extended beyond natural datasets to encompass a spectrum of scenarios, encompassing mixed noisy datasets (comprising both natural and noisy data) as well as fairness-oriented datasets. This approach allowed us to substantiate the efficacy of MLPrior across various contexts.

7 Related Work

7.1 Test Prioritization Techniques

Test prioritization aims to establish an optimized sequencing of tests with the objective of early detection of system bugs. In the field of traditional software testing, numerous test prioritization approaches have been proposed [104], [105], [106], [107], [108]. Lou *et al.* [109] introduced an innovative approach to prioritize test cases, focusing on the inherent ability of individual test cases to detect faults. Their approach consists of two distinct models: a statistics-based model and a probability-based model, both of which quantify the fault detection capability of each test case. Through empirical evaluations, they demonstrated that the statistics-based model outperformed alternative methods, underscoring the significance of incorporating fault detection capability within the realm of test case prioritization. Henard *et al.* [110] conducted a thorough comparative study to analyze existing test prioritization techniques, finding that the difference between white-box strategies [111] and black-box strategies [112] are small. Chen *et al.* [113], in pursuit of enhancing the velocity of compiler testing, introduced the LET (Learning and Scheduling-based Test prioritization) framework. This pioneering framework is underpinned by two salient processes: the learning process, designed to discern program features and prognosticate the potential of a novel test program in revealing bugs, and the scheduling process, which strategically prioritizes test programs based on their propensity to unveil bugs.

In addition to the traditional field of software engineering, multiple test input prioritization strategies have been proposed in the literature for Deep Neural Networks (DNNs) [114], [15], [16], [20] to tackle the labeling-cost issue. Feng *et al.* [16] introduced DeepGini, which prioritizes tests by utilizing the Gini score to measure model confidence for each test input. Byun *et al.* [115] assessed various white-box metrics for ranking bug-revealing inputs, encompassing widely-used measures like softmax confidence, Bayesian uncertainty, and input surprise. Furthermore, Weiss *et al.* [20] extensively investigated diverse test input prioritization techniques for DNNs, particularly focusing on uncertainty-based metrics such as Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy. These metrics have demonstrated effectiveness in identifying potentially misclassified test inputs and have played a crucial role in facilitating test prioritization endeavors. Furthermore, Wang *et al.* [15] proposed a mutation-based test prioritization approach for DNNs, which will be described in the subsequent Section 7.4.

7.2 DNN Testing

In addition to test prioritization, the domain of DNN testing encompasses several other pivotal areas, such as test selection [55], [56], test input generation [17], [116], and test adequacy. Test selection aims to select a representative subset from the original test set to estimate the accuracy of the entire test set. Various test selection approaches have been

proposed in the literature. Li *et al.* [56] proposed CES (Cross Entropy-based Sampling), which performs test selection by minimizing the cross-entropy between the selected test set and the entire test set, ensuring that the distribution of the selected test set closely matches the original set. Chen *et al.* [55] proposed PACE, which selected representative test inputs based on clustering, prototype selection, and adaptive random testing. First, Pace divides all test inputs into clusters based on their testing capabilities. Then, PACE utilizes the MMD-critic algorithm [33] to select prototypes from each group. For tests not belonging to any groups, PACE leverages adaptive random testing [117] to select test inputs by considering diversity.

Within the domain of test input generation, researchers have proposed a multitude of techniques aimed at generating diverse and effective inputs for DNN systems. Pei *et al.* [17] proposed DeepXplore, a white-box differential technique that focuses on generating test inputs capable of effectively evaluating the robustness of real-world DL systems. By leveraging the notion of neuron coverage, DeepXplore generates inputs that cover distinct regions of the neural network. Tian *et al.* [116] presented DeepTest, a method specifically tailored for generating test inputs to assess the performance of autonomous driving systems. DeepTest employs a greedy search strategy in conjunction with nine realistic image transformations to produce a diverse set of challenging input data. By systematically exploring the input space, DeepTest aims to uncover potential failures or limitations in autonomous driving systems, thereby enhancing their safety and reliability.

Regarding test adequacy, Ma *et al.* [18] proposed a set of multi-granularity testing criteria, including k-multisection neuron coverage, neuron boundary coverage, and strong neuron activation coverage. These criteria have been developed to identify corner behaviors and uncover potential vulnerabilities in DNN systems by comprehensively examining the coverage of various aspects of the neural network's behavior. Kim *et al.* [118] introduced surprise adequacy as a novel test adequacy criterion for testing DL systems. The surprise adequacy criterion emphasizes the importance of a test input being both challenging and informative while still adhering reasonably to the underlying training data distribution. This criterion emphasizes that a good test input should be sufficiently challenging and informative but should not deviate excessively from the training data distribution.

7.3 Mutation-based Test Prioritization for Traditional Software

Mutation testing [63] entails generating intentional defects, referred to as mutants, within the software code to assess the test suite's quality. In the field of traditional software testing [109], [119], [23], mutation testing can be employed to assess the fault-detection capabilities of individual test cases, thereby achieving test prioritization. Lou *et al.* [109] introduced a novel test-case prioritization approach that determines the order of test cases by considering their fault detection ability. This ability is defined based on the analysis of mutation faults simulated from real software faults. By strategically ordering the test cases, this approach aims to maximize the efficiency of the testing process by prioritizing the detection of critical faults. Papadakis *et al.* [23] conducted a mutation analysis as an alternative technique to Combinatorial Interaction Testing (CIT). Their research suggests that the mutants generated using their approach demonstrate a stronger correlation with code-level faults than the input interactions targeted by the CIT approach. This underscores the potential of mutation analysis to offer valuable insights into underlying faults within software systems and guide test case prioritization. Furthermore, Shin *et al.* [119] proposed a novel test case prioritization method that combines mutation-based and diversity-based approaches. They demonstrate that mutation-based prioritization is as effective as, or more effective than, random prioritization and coverage-based prioritization.

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

Mutation Testing for DNNs The field of mutation testing for DNNs has seen significant exploration, with numerous studies contributing to the evolution of various mutation operators and frameworks [24], [120], [114]. A notable contribution in this domain is from Ma *et al.* [24], who introduced DeepMutation. This innovative approach is designed to assess the quality of test data for DL systems through comprehensive mutation testing. DeepMutation encompasses a diverse array of mutation operators at both the source and model levels. These operators are meticulously crafted to inject faults into different components of DL systems, including training data, programming code, and the models themselves. Building upon this foundation, Hu *et al.* further expanded their work with the development of DeepMutation++ [120], an advanced mutation testing tool specifically tailored for DL systems. DeepMutation++ introduced a set of new mutation operators that are particularly suited for feed-forward neural networks (FNNs) and Recurrent Neural Networks (RNNs). A key feature of this tool is its capability to dynamically mutate the runtime states of RNNs, a critical aspect for evaluating the resilience of these networks under various operational conditions. Humbatova *et al.* [114] made a significant stride in the field by developing DeepCrime, the first mutation testing tool that implements DL mutation operators grounded in actual DL faults. DeepCrime is characterized by its comprehensive set of 24 newly defined mutation operators. These operators are not just theoretical constructs but are based on real-world faults observed in DL systems, making DeepCrime a highly practical tool for testing and improving the reliability of these systems.

Mutation-based test prioritization for DNNs Wang *et al.* [15] introduced PRIMA, an innovative test input prioritization technique founded on intelligent mutation analysis. PRIMA is applicable to both classification and regression models and possesses the capability to handle test inputs generated through adversarial input generation techniques, thereby enhancing the probability of misclassification. However, PRIMA's model mutation rules cannot be adapted to classical ML models.

In this study, we proposed MLPrior, a mutation-based test input prioritization approach specifically designed for

classical ML models. The significant differences between MLPrior and PRIMA are as follows:

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

8 Conclusion

In order to solve the labeling cost problem for classical ML models, we propose MLPrior, which prioritizes tests that are more likely to be misclassified. MLPrior leverages the unique characteristics of classical ML classifiers, including their interpretability and carefully engineered dataset features, to effectively prioritize test inputs. The foundational principles of MLPrior are twofold: Firstly, tests exhibiting higher sensitivity to mutations are more likely to be misclassified. Secondly, tests situated closer to the decision boundary of the model are more susceptible to misclassification. Capitalizing on these principles, we design mutation rules specifically for classical ML models and their datasets. For each test, we generate mutation features while simultaneously transforming its attribute into a feature vector that can indirectly quantify the proximity between it and the decision boundary. Concatenating these features, MLPrior constructs a final vector for each test, which will be inputted into a pre-trained ranking model for the purpose of predicting its misclassification probability. Finally, MLPrior ranks all the tests according to their misclassification scores in descending order. We conducted an extensive study to evaluate MLPrior, utilizing 185 different types of subjects that encompass natural, noisy, and fairness datasets. The experimental results demonstrate that MLPrior exhibits higher effectiveness compared to existing test prioritization methods, yielding an average improvement of 14.74%∼66.93% on natural datasets, 18.55%∼67.73% on mixed noisy datasets, and 15.34%∼62.72% on fairness datasets.

Availability

Our replication package is available at

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

References

[1] 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.

[2] 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*, 2016, pp. 770–778.

[3] D. W. Otter, J. R. Medina, and J. K. Kalita, "A survey of the usages of deep learning for natural language processing," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 2, pp. 604–624, 2020.

[4] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, 2020, pp. 38–45.

[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] J. Zeng, H. Tang, Y. Li, and X. He, "A deep learning model based on sparse matrix for point-of-interest recommendation." in *SEKE*, 2019, pp. 379–492.

[7] D. V. Carvalho, E. M. Pereira, and J. S. Cardoso, "Machine learning interpretability: A survey on methods and metrics," *Electronics*, vol. 8, no. 8, p. 832, 2019.

[8] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[9] Y.-Y. Song and L. Ying, "Decision tree methods: applications for classification and prediction," *Shanghai archives of psychiatry*, vol. 27, no. 2, p. 130, 2015.

[10] R. E. Wright, "Logistic regression." 1995.

[11] 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.

[12] M. J. Raihan, M. A.-M. Khan, S.-H. Kee, and A.-A. Nahid, "Detection of the chronic kidney disease using xgboost classifier and explaining the influence of the attributes on the model using shap," *Scientific Reports*, vol. 13, no. 1, p. 6263, 2023.

[13] D. Yu, Z. Liu, C. Su, Y. Han, X. Duan, R. Zhang, X. Liu, Y. Yang, and S. Xu, "Copy number variation in plasma as a tool for lung cancer prediction using extreme gradient boosting (xgboost) classifier," *Thoracic cancer*, vol. 11, no. 1, pp. 95–102, 2020.

[14] T. W. Cenggoro, B. Mahesworo, A. Budiarto, J. Baurley, T. Suparyanto, and B. Pardamean, "Features importance in classification models for colorectal cancer cases phenotype in indonesia," *Procedia Computer Science*, vol. 157, pp. 313–320, 2019.

[15] 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)*. IEEE, 2021, pp. 397–409.

[16] 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*, 2020, pp. 177–188.

[17] 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*, 2017, pp. 1–18.

[18] 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*, 2018, pp. 120–131.

[19] M. Wicker, X. Huang, and M. Kwiatkowska, "Feature-guided black-box safety testing of deep neural networks," in *Tools and Algorithms for the Construction and Analysis of Systems: 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I 24*. Springer, 2018, pp. 408–426.

[20] 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*, 2022, pp. 139–150.

[21] B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran, "Killing strategies for model-based mutation testing," *Softw. Test. Verification Reliab.*, vol. 25, no. 8, pp. 716–748, 2015. [Online]. Available: https://doi.org/10.1002/stvr.1522

[22] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P. Schobbens, and P. Heymans, "Featured model-based mutation analysis," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 655–666. [Online]. Available: https://doi.org/10.1145/2884781.2884821

[23] 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*. IEEE Computer Society, 2014, pp. 1–10. [Online]. Available: https://doi.org/10.1109/ICST.2014.11

[24] 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)*. IEEE, 2018, pp. 100–111.

[25] X. Gao, J. Zhai, S. Ma, C. Shen, Y. Chen, and Q. Wang, "Fairneuron: improving deep neural network fairness with adversary games on selective neurons," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 921–933.

[26] Z. Chen, J. M. Zhang, F. Sarro, and M. Harman, "Maat: a novel ensemble approach to addressing fairness and performance bugs for machine learning software," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1122–1134.

[27] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yogamani, and P. Pérez, "Deep reinforcement learning for autonomous driving: A survey," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 6, pp. 4909–4926, 2021.

[28] M. Fatima, M. Pasha *et al.*, "Survey of machine learning algorithms for disease diagnostic," *Journal of Intelligent Learning Systems and Applications*, vol. 9, no. 01, p. 1, 2017.

[29] 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.

[30] S. Mallat, "Understanding deep convolutional networks," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2065, p. 20150203, 2016.

[31] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*. Springer, 2014, pp. 818–833.

[32] B. Yeo and D. Grant, "Predicting service industry performance using decision tree analysis," *International Journal of Information Management*, vol. 38, no. 1, pp. 288–300, 2018.

[33] 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.

[34] M. Sewak, S. K. Sahay, and H. Rathore, "Comparison of deep learning and the classical machine learning algorithm for the malware detection," in *2018 19th IEEE/ACIS international conference on software engineering, artificial intelligence, networking and parallel/distributed computing (SNPD)*. IEEE, 2018, pp. 293–296.

[35] M. Ghassemi, T. Naumann, P. Schulam, A. L. Beam, I. Y. Chen, and R. Ranganath, "A review of challenges and opportunities in machine learning for health," *AMIA Summits on Translational Science Proceedings*, vol. 2020, p. 191, 2020.

[36] F. Rundo, F. Trenta, A. L. di Stallo, and S. Battiato, "Machine learning for quantitative finance applications: A survey," *Applied Sciences*, vol. 9, no. 24, p. 5574, 2019.

[37] P. Weber, K. V. Carl, and O. Hinz, "Applications of explainable artificial intelligence in finance—a systematic review of finance, information systems, and computer science literature," *Management Review Quarterly*, pp. 1–41, 2023.

[38] C. Chen, K. Lin, C. Rudin, Y. Shaposhnik, S. Wang, and T. Wang, "A holistic approach to interpretability in financial lending: Models, visualizations, and summary-explanations," *Decision Support Systems*, vol. 152, p. 113647, 2022.

[39] A. Adadi and M. Berrada, "Explainable ai for healthcare: from black box to interpretable models," in *Embedded Systems and Artificial Intelligence: Proceedings of ESAI 2019, Fez, Morocco*. Springer, 2020, pp. 327–337.

[40] M. Verdicchio and A. Perin, "When doctors and ai interact: on human responsibility for artificial risks," *Philosophy & Technology*, vol. 35, no. 1, p. 11, 2022.

[41] H. Smith, "Clinical ai: opacity, accountability, responsibility and liability," *Ai & Society*, vol. 36, no. 2, pp. 535–545, 2021.

[42] J. Amann, A. Blasimme, E. Vayena, D. Frey, V. I. Madai, and P. Consortium, "Explainability for artificial intelligence in healthcare: a multidisciplinary perspective," *BMC medical informatics and decision making*, vol. 20, pp. 1–9, 2020.

[43] T. Grote and P. Berens, "On the ethics of algorithmic decision-making in healthcare," *Journal of medical ethics*, 2019.

[44] H. Yan, S. Lin *et al.*, "New trend in fintech: Research on artificial intelligence model interpretability in financial fields," *Open Journal of Applied Sciences*, vol. 9, no. 10, p. 761, 2019.

[45] K. Suzuki, "Overview of deep learning in medical imaging," *Radiological physics and technology*, vol. 10, no. 3, pp. 257–273, 2017.

[46] D. Shen, G. Wu, and H.-I. Suk, "Deep learning in medical image analysis," *Annual review of biomedical engineering*, vol. 19, pp. 221–248, 2017.

[47] Z. A. Shirazi, C. P. de Souza, R. Kashef, and F. F. Rodrigues, "Deep learning in the healthcare industry: theory and applications," in *Computational intelligence and soft computing applications in healthcare management science*. IGI Global, 2020, pp. 220–245.

[48] R. Shwartz-Ziv and A. Armon, "Tabular data: Deep learning is not all you need," *Information Fusion*, vol. 81, pp. 84–90, 2022.

[49] Y. Wang and T. Wang, "Application of improved lightgbm model in blood glucose prediction," *Applied Sciences*, vol. 10, no. 9, p. 3227, 2020.

[50] L. H. Gilpin, D. Bau, B. Z. Yuan, A. Bajwa, M. Specter, and L. Kagal, "Explaining explanations: An overview of interpretability of machine learning," in *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*. IEEE, 2018, pp. 80–89.

[51] M. A. Hanif, F. Khalid, R. V. W. Putra, S. Rehman, and M. Shafique, "Robust machine learning systems: Reliability and security for deep neural networks," in *2018 IEEE 24th international symposium on on-line testing and robust system design (IOLTS)*. IEEE, 2018, pp. 257–260.

[52] N. Mehrabi, F. Morstatter, N. Saxena, K. Lerman, and A. Galstyan, "A survey on bias and fairness in machine learning," *ACM Computing Surveys (CSUR)*, vol. 54, no. 6, pp. 1–35, 2021.

[53] A. N. Bhagoji, D. Cullina, C. Sitawarin, and P. Mittal, "Enhancing robustness of machine learning systems via data transformations," in *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*. IEEE, 2018, pp. 1–5.

[54] 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.

[55] 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.

[56] 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*, 2019, pp. 499–509.

[57] 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.

[58] 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.

[59] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant, *Applied logistic regression*. John Wiley & Sons, 2013.

[60] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and regression trees*. Routledge, 2017.

[61] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978. [Online]. Available: https://doi.org/10.1109/C-M.1978.218136

[62] P. Ammann and J. Offutt, *Introducation to Software Testing*. Cambridge University Press, 2008.

[63] 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.

[64] 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)*. IEEE, 2020, pp. 74–84.

[65] 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.

[66] G. Petrovic, M. Ivankovic, B. Kurtz, P. Ammann, and R. Just, "An industrial application of mutation testing: Lessons, challenges, and research directions," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018, pp. 47–53.

[67] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 2, pp. 99–118, 1996.

[68] 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*, 2009, pp. 297–298.

[69] 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*. Elsevier, 2019, vol. 112, pp. 275–378.

[70] T. Fredriksson, J. Bosch, and H. H. Olsson, "Machine learning models for automatic labeling: A systematic literature review." *ICSOFT*, pp. 552–561, 2020.

[71] T. Fredriksson, D. I. Mattos, J. Bosch, and H. H. Olsson, "Data labeling: An empirical investigation into industrial challenges and mitigation strategies," in *International Conference on Product-Focused Software Process Improvement*. Springer, 2020, pp. 202–216.

[72] M. Desmond, E. Duesterwald, K. Brimijoin, M. Brachman, and Q. Pan, "Semi-automated data labeling," in *NeurIPS 2020 Competition and Demonstration Track*. PMLR, 2021, pp. 156–169.

[73] J. Wu, C. Ye, V. S. Sheng, Y. Yao, P. Zhao, and Z. Cui, "Semi-automatic labeling with active learning for multi-label image classification," in *Advances in Multimedia Information Processing–PCM 2015: 16th Pacific-Rim Conference on Multimedia, Gwangju, South Korea, September 16-18, 2015, Proceedings, Part I 16*. Springer, 2015, pp. 473–482.

[74] "Making automated data labeling a reality in modern ai," *Accessed*, 2023. [Online]. Available: https://www.snorkel.ai/blog/automated-data-labeling

[75] G. Guo, H. Wang, D. Bell, Y. Bi, and K. Greer, "Knn model-based approach in classification," in *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003. Proceedings*. Springer, 2003, pp. 986–996.

[76] H. Kim and Z. Gu, "A logistic regression analysis for predicting bankruptcy in the hospitality industry," *The Journal of Hospitality Financial Management*, vol. 14, no. 1, pp. 17–34, 2006.

[77] A. Mayr, H. Binder, O. Gefeller, and M. Schmid, "The evolution of boosting algorithms," *Methods of information in medicine*, vol. 53, no. 06, pp. 419–427, 2014.

[78] P. Chen, S. Liu, H. Zhao, and J. Jia, "Gridmask data augmentation," *arXiv preprint arXiv:2001.04086*, 2020.

[79] 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.

[80] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[81] S. Boughorbel, F. Jarray, and M. El-Anbari, "Optimal classifier for imbalanced data using matthews correlation coefficient metric," *PloS one*, vol. 12, no. 6, p. e0177678, 2017.

[82] "The adult census income dataset," 2017. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/adult

[83] "The bank dataset." 2014. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Bank+Marketing

[84] T. Tazin, M. N. Alam, N. N. Dola, M. S. Bari, S. Bourouis, M. Monirujjaman Khan *et al.*, "Stroke disease detection and prediction using robust learning approaches," *Journal of healthcare engineering*, vol. 2021, 2021.

[85] G. ÖZSEZER and G. MERMER, "Diabetes risk prediction with machine learning models," *Artificial Intelligence Theory and Applications*, vol. 2, no. 2, pp. 1–9, 2022.

[86] J. Hua, B. Chu, J. Zou, and J. Jia, "Ecg signal classification in wearable devices based on compressed domain," *Plos one*, vol. 18, no. 4, p. e0284008, 2023.

[87] S. Tizpaz-Niari, A. Kumar, G. Tan, and A. Trivedi, "Fairness-aware configuration of machine learning libraries," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 909–920.

[88] Y. Li, L. Meng, L. Chen, L. Yu, D. Wu, Y. Zhou, and B. Xu, "Training data debugging for the fairness of machine learning software," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2215–2227.

[89] H. Zheng, Z. Chen, T. Du, X. Zhang, Y. Cheng, S. Ji, J. Wang, Y. Yu, and J. Chen, "Neuronfair: Interpretable white-box fairness testing through biased neuron identification," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1519–1531.

[90] D. Dua and C. Graff, "Uci machine learning repository. university of california, school of information and computer science, irvine, ca (2019)," 2019.

[91] R. Kohavi *et al.*, "Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid." in *Kdd*, vol. 96, 1996, pp. 202–207.

[92] A. Ogunleye and Q.-G. Wang, "Xgboost model for chronic kidney disease diagnosis," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 17, no. 6, pp. 2131–2140, 2019.

[93] C. Rao, Y. Liu, and M. Goh, "Credit risk assessment mechanism of personal auto loan based on pso-xgboost model," *Complex & Intelligent Systems*, vol. 9, no. 2, pp. 1391–1414, 2023.

[94] M. Mukid, T. Widiharih, A. Rusgiyono, and A. Prahutama, "Credit scoring analysis using weighted k nearest neighbor," in *Journal of Physics: Conference Series*, vol. 1025, no. 1. IOP Publishing, 2018, p. 012114.

[95] H. Kamel, D. Abdulah, and J. M. Al-Tuwaijari, "Cancer classification using gaussian naive bayes algorithm," in *2019 International Engineering Conference (IEC)*. IEEE, 2019, pp. 165–170.

[96] D. Slack, S. A. Friedler, C. Scheidegger, and C. D. Roy, "Assessing the local interpretability of machine learning models," *arXiv preprint arXiv:1902.03501*, 2019.

[97] Y. Li, J. Wang, and C. Wang, "Systematic testing of the data-poisoning robustness of knn," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023.

[98] S. A. Friedler, C. Scheidegger, S. Venkatasubramanian, S. Choudhary, E. P. Hamilton, and D. Roth, "A comparative study of fairness-enhancing interventions in machine learning," in *Proceedings of the conference on fairness, accountability, and transparency*, 2019, pp. 329–338.

[99] A. Stevens, P. Deruyck, Z. Van Veldhoven, and J. Vanthienen, "Explainability and fairness in machine learning: Improve fair end-to-end lending for kiva," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2020, pp. 1241–1248.

[100] R. J. Lewis, "An introduction to classification and regression tree (cart) analysis," in *Annual meeting of the society for academic emergency medicine in San Francisco, California*, vol. 14. Citeseer, 2000.

[101] 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.

[102] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[103] M. Fan, W. Wei, W. Jin, Z. Yang, and T. Liu, "Explanation-guided fairness testing through genetic algorithm," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 871–882.

[104] 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.

[105] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse, "Adaptive random test case prioritization," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 233–244.

[106] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei, "Prioritizing junit test cases in absence of coverage information," in *2009 IEEE International Conference on Software Maintenance*. IEEE, 2009, pp. 19–28.

[107] 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*. IEEE, 2006, pp. 123–133.

[108] 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*, 2009, pp. 201–212.

[109] 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)*. IEEE, 2015, pp. 46–57.

[110] 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)*. IEEE, 2016, pp. 523–534.

[111] 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.

[112] 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.

[113] 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.

[114] 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*, 2021, pp. 67–78.

[115] 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)*. IEEE, 2019, pp. 63–70.

[116] 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*, 2018, pp. 303–314.

[117] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making: 9th Asian Computing Science Conference. Dedicated to Jean-Louis Lassez on the Occasion of His 5th Birthday. Chiang Mai, Thailand, December 8-10, 2004. Proceedings 9*. Springer, 2005, pp. 320–329.

[118] 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)*. IEEE, 2019, pp. 1039–1049.

[119] 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.

[120] 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)*. IEEE, 2019, pp. 1158–1161.