Robust Test Selection for Deep Neural Networks

Weifeng Sun, Meng Yan, Zhongxin Liu, David Lo

Abstract—Deep Neural Networks (DNNs) have been widely used in various domains, such as computer vision and software engineering. Although many DNNs have been deployed to assist various tasks in the real world, similar to traditional software, they also suffer from defects that may lead to severe outcomes. DNN testing is one of the most widely used methods to ensure the quality of DNNs. Such method needs rich test inputs with oracle information (expected output) to reveal the incorrect behaviors of a DNN model. However, manually labeling all the collected test inputs is a labor-intensive task, which delays the quality assurance process. *Test selection* tackles this problem by carefully selecting a small, more suspicious set of test inputs to label, enabling the failure detection of a DNN model with reduced effort. Researchers have proposed different test selection methods, including neuron-coverage-based and uncertainty-based methods, where the uncertainty-based method is arguably the most popular technique. Unfortunately, existing uncertainty-based selection methods meet the performance bottleneck due to one or several limitations: 1) they ignore noisy data in real scenarios; 2) they wrongly exclude many *failure-revealing test inputs* but rather include many *successful test inputs* (referring to those test inputs that are correctly predicted by the model); 3) they ignore the diversity of the selected test set. In this paper, we propose RTS, a **R**obust **T**est **S**election method for deep neural networks to overcome the limitations mentioned above. First, RTS divides all unlabeled candidate test inputs into noise set, successful set, and suspicious set and assigns different selection prioritization to divided sets, which effectively alleviates the impact of noise and improves the ability to identify suspect test inputs. Subsequently, RTS leverages a probability-tier-matrix-based test metric for prioritizing the test inputs in each divided set (i.e., suspicious, successful, and noise set). As a result, RTS can select more suspicious test inputs within a limited selection size. We evaluate RTS by comparing it with 14 baseline methods under 5 widely-used DNN models and 6 widely-used datasets. The experimental results demonstrate that RTS can significantly outperform all test selection methods in failure detection capability and the test suites selected by RTS have the best model optimization capability. For example, when selecting 2.5% test input, RTS achieves an improvement of 9.37%-176.75% over baseline methods in terms of failure detection.

Index Terms—Deep learning testing, deep neural networks, test selection.

1 Introduction

Deep Neural Networks (DNNs) are increasingly applied to solving many complex problems in various applications, such as code-related tasks (e.g., code summarization [1] and code search [2]), face recognition [3], and autonomous vehicles [4]. Similar to traditional software, DNNs also contain defects that can lead to severe outcomes, e.g., autonomous vehicle accidents [5]. Therefore, guaranteeing the quality of DNNs becomes imperative and critical.

In the initial engineering phase, data scientists typically acquire a substantial dataset from the target distribution. They label the data, divide it into training, validation, and test sets, and evaluate the DNN's performance on the test set (referred to as the *original test set* hereafter). However, relying solely on this original test set is insufficient to ensure the quality of the DNN. During training, the model may mistakenly learn repetitive data patterns [6]. To address this issue, frequent and high-quality DNN testing is necessary to identify any mislearned attributes before deployment. DNN testing entails generating or gathering additional test data and labeling them. Unfortunately, collecting labeled test inputs is often expensive and time-consuming, because: 1) The manual labeling process requires a lot of human resources and time costs [7]. 2) In specific applications, such as protein structure prediction [8], the labeling process requires people to have domain-specific knowledge. 3) The test set is large, but *failure-revealing test inputs* that can trigger the system's potential errors usually account for only a few [9]–[11].

To relieve this problem, *test selection* (for short, **TS**) has emerged to reduce labeling effort. Test selection is designed to address two common problems: 1) selecting data that can be used to estimate model performance and represent the entire set; 2) selecting data that are more likely to be incorrectly classified by the model and then retraining a better model with the selected data. This paper focuses on the second problem. Given a DNN under test and a large set of unlabelled test inputs (denoted as *candidate set*), TS utilizes specific information, such as internal state or output result of the model, to select some test inputs so that those test inputs that are likely to expose failures in a DNN can be labeled earlier. As the selected test set is usually small-scale and detects failures in the model, it can significantly reduce the labeling overhead and improve the efficiency of DNN testing. Subsequently, testers can label selected inputs and use them to retrain the model, thereby optimizing the model's performance. A satisfactory TS method needs to satisfy the following two conditions: 1) The selected inputs can detect many and diverse failures of the DNN model; 2) The selected inputs can improve/optimize the performance of the DNN model as much as possible. However, how to select representative and diverse inputs is still an open

- *Weifeng Sun, Meng Yan is with the School of Big Data and Software Engineering, Chongqing University, China. E-mail: weifeng.sun@cqu.edu.cn, mengy@cqu.edu.cn*
- *Zhongxin Liu is with the State Key Laboratory of Blockchain and Data Security and the College of Computer Science and Technology, Zhejiang University, China. E-mail: liu_zx@zju.edu.cn*
- *David Lo is with the School of Information Systems, Singapore Management University, Singapore. E-mail: davidlo@smu.edu.sg*

Meng Yan is the corresponding author.

question.

To date, researchers have proposed different test selection methods, including neuron-coverage-based and uncertainty-based methods [12]–[15]. Inspired by the success of using code coverage to facilitate traditional software testing, testing criteria based on *neuron coverage* [9], [10], [16], such as neuron activation coverage [9] and neuron boundary coverage [10], have been proposed to verify the adequacy of testing. However, there are plenty of discussions on their failure detection performance and usage scenarios [12], [17], because: 1) collecting neuron activation states could be very expensive, especially for complex DNNs; 2) achieving the maximum coverage for most of existing neuron coverage criteria is a trivial task. For example, Feng et al. [12] point out that only using about 1% of tests in a test set could achieve the maximum coverage for top-k neuron coverage criterion [10].

Additionally, uncertainty-based TS methods, such as DeepGini [12] and ATS [15], select test inputs by estimating the *uncertainty* of the DNN on the reported classification probability. Lower uncertainty means that the model has high confidence for the prediction results. The uncertainty-based TS method is arguably the most popular technique adopted by test selection [12], [15], [18] and have been demonstrated to be more effective than neuron-coverage-based ones [12], [18] in detecting failures. Despite that, they also have limited application scenarios. For example, when the training data are contaminated or transfer learning is used for training, the performance of DeepGini largely degrades [19]. We review the state-of-the-art uncertainty-based TS methods and find their effectiveness bottlenecks are due to one or several of the following issues:

(1) **Ignoring noisy data in real-world scenarios.** *Failure-revealing test inputs are the test inputs that are valid and can trigger the model's wrong behavior*. To evaluate the effectiveness of TS method, existing works select test inputs under a constructed candidate set that contains the original test inputs and their variants generated using data mutation operations [20]. However, in real-world scenarios, the candidate test inputs, such as document data sets [21], Web data [22], [23], and image data [24] usually contain some invalid data, i.e., noise [25]. For example, if the task is to classify handwritten digits, printed digits can be considered as noise [26]. Noise is "irrelevant or meaningless data" [25] and can significantly hinder testing analysis and waste the manual inspection effort of testers. Unfortunately, existing TS methods are noise-sensitive and do not explicitly handle it, resulting in fewer failure-revealing test inputs being identified.

(2) **Omission of failure-revealing test inputs and inclusion of successful test inputs.** Generally, the uncertainty-based TS method selects test inputs near the decision boundary of the DNN model by estimating their classification uncertainty. Nevertheless, such selection has two limitations: 1) selecting some *successful test inputs* (referring to those test inputs that are correctly predicted by the model) near the decision boundary; 2) failing to select some failure-revealing test inputs far from the decision boundary.

(3) **Ignoring the diversity of the selected test set.** In software testing, testers always seek failure behavior diversity so that test cases can trigger different types of failures for a program under given test resources. We also expect that TS should reveal as diverse failures of the DNN model as possible for better testing analysis. However, many existing TS methods only focus on the percentage of the selected failure-revealing test inputs, ignoring their diversity.

To alleviate the above problems, we explore an alternative TS method to boost the testing of DNN models. First, to address **issue 1**, considering noisy data in real scenarios, an intuitive idea is to explicitly identify and filter out noisy data before test selection. Based on this idea, we employ a retrieval technique to search the k most similar training data for each test input. Those test inputs which are not similar to their retrieved neighbors will be regarded as noisy data. Next, we find that **issue 2** happens because uncertainty-based TS methods only consider the uncertainty of the test inputs and ignore the relationship between them, i.e., the class labels to which similar instances belong. A well-trained DNN model needs to classify the test inputs and their similar instances to the same class. For this reason, we retrieve each test input's similar instances and utilize majority voting [27] to estimate the label of test input based on retrieved instances. We effectively distinguish suspicious and successful test inputs by checking the consistency between the estimated and predicted labels. Through the above-mentioned process, we have divided the candidate set into noise, successful, and suspicious sets. To alleviate **issue 3**, it would be better to select the test inputs that differ from each other to achieve test set diversity. The faults of model often propagate to its observable output, thereby affecting user decision-making. Intuitively, the complexity and richness of the test output can increase the likelihood of fault propagation, making the test criteria more effective. Inspired by this, a novel test metric measuring the difference between test inputs in output probabilities is applied to each divided set to guide test selection. Given that the output vector is continuous, we introduce the concept of *probability tier matrix* that transforms the continuous prediction probability of test input into a discrete sequence of level indexes of probability tier matrix. Further, we propose a fitness metric to measure the difference between a test input and a set of selected test inputs. Based on these approaches, we propose a **R**obust **T**est **S**election (RTS) method for deep neural networks. RTS can select a subset of test inputs without class labels, revealing more and diverse failures in the DNN model and reducing the labeling effort for the optimization process.

We conduct extensive experiments with five well-designed DNN models and six widely-used datasets to evaluate the effectiveness of RTS and compare it with 14 baseline methods. We consider different pollution scenarios in realistic unfiltered datasets and simulate them in our experiments. The experimental results demonstrate that RTS significantly outperforms all baseline methods in failure detection. Further, we prove that the test inputs selected by RTS are more effective in enhancing the performance of the DNN model than other TS methods, such as neuron-coverage-based methods and the state-of-the-art methods DeepGini and ATS.

The contributions of this paper could be summarized as follows:

- **Approach.** We propose a robust test selection technology for DNN testing, namely RTS. Considering noisy

data in real scenarios, we propose a retrieval-based filtering algorithm to filter noisy data, which enhances the robustness of RTS in complex testing scenarios. Besides, given existing uncertainty-based TS methods are difficult to handle successful test inputs near the decision boundary and failure-revealing test inputs far from the decision boundary, we further utilize the majority voting mechanism to identify suspicious test inputs based on class labels of similar instances of test inputs, thereby improving the effectiveness of RTS. Finally, we introduce the probability tier matrix and propose a novel test metric quantifying the difference of test inputs. RTS can help the testers and programmers automatically select the test inputs that are likely to expose errors from massive unfiltered and unlabeled data, making it more efficient and economical to test a DNN model.

- **Study.** We conduct an extensive experiment to investigate the performance of RTS with 14 baseline methods, including neuron-coverage-based test selection methods and the state-of-the-art DeepGini and ATS, etc. The results show that RTS significantly outperforms other test selection methods. For example, when selecting 2.5% test input, RTS achieves an improvement of 9.37%-176.75% over other baseline methods in terms of failure detection. Additionally, the test inputs selected by RTS can efficiently enhance the performance of the DNN model.
- **Package.** We have released the source code of our test selection method and the experimental datasets online[1] to support results verification and follow-up research comparison.

The rest of this paper is organized as follows. In Section 2, we introduce some background knowledge of this work. In Section 3, we conduct some experiments, whose results motivate us to propose RTS. In Section 4, we present a detailed description of our algorithm RTS. In Section 5, we present our experimental settings, including studied datasets and models, candidate test input construction, baseline methods, research questions, and so on. Section 6 reports experimental results and discoveries. Section 7 discusses various aspects related to the performance of RTS. These include its effectiveness in identifying noise, its performance in noiseless candidate sets, the setting of parameters, its performance in adversarial examples, as well as its advantages and limitations. Sections 8 and 9 describe threats to validity and related works, respectively. Section 10 presents the conclusion of our paper and discuss future work.

2 Background

This section introduces the preliminary knowledge of DNN, DNN testing, and existing TS methods, including neuron-coverage-based and uncertainty-based metrics.

2.1 The Architecture of Deep Neural Network

A Deep Neural Network (DNN) consists of multiple layers, including an input layer, one or more hidden layers, and an output layer. Each layer is made up by a series of neurons (computation unit) and neurons between adjacent layers are linked with weighted edges. Each neuron accepts the output value of the previous layer and the weights of the incoming edges and applies an activation function for nonlinear projection. The computed result is passed to the next layer through the outgoing edges. Considering a DNN model with L hidden layers and input data x, the output result y is computed as follows:

$$y = f(x) = f\left[h^{(L+1)}\left(a^{(L)}\left(h^{(L)}\left(...\left(a^{(1)}\left(h^{(1)}(x)\right)\right)\right)\right)\right)\right] \tag{1}$$

where $a^{(i)}$ refers to the hidden-layer activation functions that often have the same form at each level, and each *preactivation function* $h^{(i)}$ is typically a linear projection with weights $\mathbf{W}^{(i)}$ and bias $\mathbf{b}^{(i)}$.

$$h^{(i)}(x) = \mathbf{W}^{(i)}x + \mathbf{b}^{(i)} \tag{2}$$

Suppose we have a m-class classifier that can classify objects into M categories. Given an input x, the DNN adjusts its internal weights to output result $y = f(x)$ such that y approximates the x's ground truth (represented by one-hot vector). Then, y is further normalized to $\langle y_1, y_2, \ldots, y_m\rangle$ through the softmax function [28], where $\sum_{i=1}^{M} y_i = 1$ and each element y_i is the corresponding probability value that x belongs to the class i. Thus the output domain Y needs to satisfy:

$$Y = \{y | y \in \mathbb{R}^m, ||y||_1 = 1 \wedge \forall i, y(i) \geq 0\} \tag{3}$$

The final output vector processed by softmax function is denoted as:

$$\mathrm{DNN}(x) = y \in Y \tag{4}$$

From now on, we denote the final output vector produced by the softmax function as *prediction probability* or *classification probability*.

Given a test input x, the DNN model classifies it into the i-th class (denoted by *prediction/classification class*), iff i-th element of y is the largest element, i.e., $i = \mathrm{argmax}_i(y_i)$. If $\max(y_i)$ is closer to 1, we consider that the model has higher confidence for the classification result of x.

2.2 Deep Learning Testing and Test Selection

DNN testing [9], [10], [29]–[33] is crucial for ensuring the robustness of DNNs before online deployment. Robustness, in essence, implies that a slight change in the input should not cause a significant change in the output of a well-trained DNN model [34], which forms the foundation of RTS. Therefore, DNN testing usually generates or collects test inputs that exhibit slight variations from the training data but still adhere to the underlying target distribution. These inputs are then used to assess the model's robustness. It is important to recognize that when a well-tested DNN model is deployed in a real application, its effectiveness may degrade over time. This decline in performance is not attributed to inherent faults within the model itself, but rather arises from the continuous influx of new data and the consequential significant drifts between the original data distribution and the distribution of unseen data, which is commonly referred to as the *domain generalization problem*. In such cases, developers may need to select a subset of

1. https://github.com/swf1996120/RTS

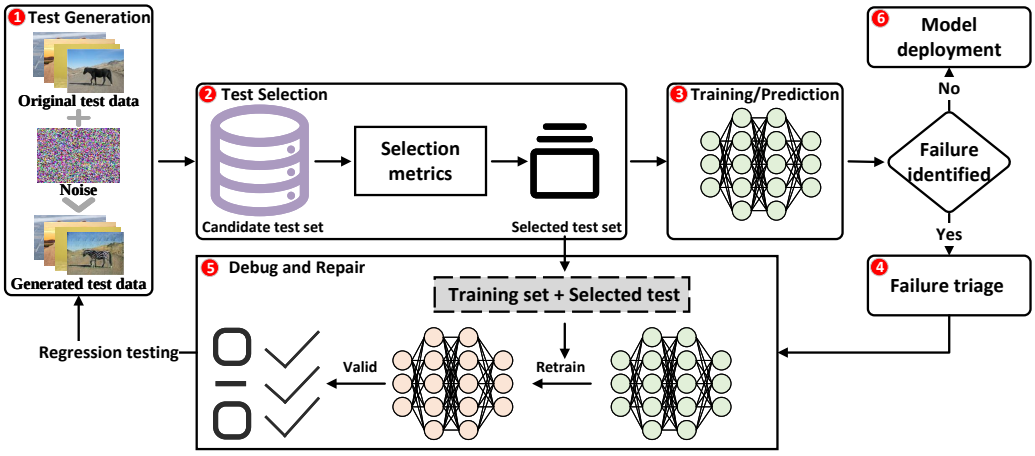


Fig. 1. Overview of **TS**.

data from the unseen data distribution to enhance the model's performance under the new domain. However, it is essential to differentiate this objective from the focus of the present paper, as domain generalization primarily focuses on enhancing the model's adaptability and generalization capabilities to effectively handle new, previously unseen domains. Conversely, DNN testing focuses on identifying and uncovering erroneous attributes that the model may have learned from historical data prior to deployment. It aims to ensure that the model's predictions are reliable and free from potential pitfalls or biases induced by the training data.

The workflow of DNN testing is depicted in Figure 1, illustrating the key steps involved. Initially, a prototype model is generated based on historical data. Developers then either sample test inputs from the collected data or generate inputs based on specific requirements [35]. These inputs form the candidate test set. Due to the vast input space, test selection techniques can be employed to identify data that are more likely to be misclassified by the model from the candidate test set. Next, developers label selected data, run a built model against them, and check the predictive accuracy. Any identified failures are labeled with different severity levels and assigned to respective developers for further analysis and resolution. Throughout the debugging and repairing process, the selected test inputs can be integrated into the training data to improve the model's correctness through retraining.

In the domain of machine learning, active learning is a well-established iterative methodology for refining a DNN over progressive stages. Each iteration involves the selection and labeling of a strategically chosen dataset to enhance the previously derived DNN. Test selection, including our proposed method, can be positioned within the active learning paradigm, functioning as a specialized adaptation. Both test selection and active learning aim to optimize DNN performance while curtailing labeling overheads. However, their goals may be slightly different. Active learning aims to select a small amount of data to train a DNN that achieves performance comparable to using the entire dataset. Conversely, test selection focuses on detecting failures in the model under test as soon as possible and fixing or improving the performance of a pre-trained DNN by retraining it with a limited amount of data. Central to both approaches is the objective to reduce the volume of labeled data.

2.3 Neuron-Coverage-Based TS Method

Neuron-coverage-based TS methods [9], [10] utilize structural neuron coverage to guide the selection process. During each iteration, such methods adopt a greedy algorithm to select the following test inputs based on the feedback from the previously selected set. Specifically, the test input to be selected needs to cover the maximum number of uncovered areas in terms of the given coverage criteria. In this section, we briefly describe several structural neuron coverage criteria.

Neuron Activation Coverage (*NAC*) is first proposed by Pei et al. [9] to discover the test inputs that trigger the DL systems to produce differential behaviors. A neuron can be considered activated, i.e., covered, when its output after passing an activation function exceeds a threshold (denoted by k). Intuitively, the more neurons are activated, the more states of DNN are explored. Thus, the rate of $NAC(k)$ can be defined:

$$NAC(k) = \frac{|\text{Activated Neurons}|}{|\text{Total Neurons}|} \tag{5}$$

k-Multisection Neuron Coverage (*KMNC(k)*) is proposed based on the *NAC*. Ma et al. further assume that instead of treating a neuron as having only two states (activated and inactivated), the output of a neuron is regarded as a range of values [10]. Specifically, assume that the output of a neuron under the training set is located in the interval $[low, high]$, and the interval is equally divided into k segments. The *KMNC(k)* attempts to make the neuron cover each segment.

Neuron Boundary Coverage (*NBC*) [10] emphasizes the coverage of test inputs across corner regions. Unlike *KMNC(k)*, *NBC* considers $(-\infty, low]$ and $[high, \infty)$ regions.

Strong Neuron Activation Coverage (*SNAC*) [10] only considers upper boundary (i.e $[high, \infty)$). Thus, we compute *SNAC* through the ratio of the neurons covering the upper bound to all the neurons.

Top-k Neuron Coverage (*TKNC(k)*) [10] counts the k most active neurons in each layer. Unlike *NAC*, a neuron can be considered activated if and only if, when running the test input on the DNN, the output of the neuron is greater than or equal to the k-th highest value in the neuron's layer. Finally,

the *TKNC(k)* is defined as the ratio of the total number of each layer's top-k neurons to the total number of neurons.

2.4 Uncertainty-Based TS Method

Uncertainty-based TS methods select test inputs for classification models by measuring the DNN's confidence in its classification results for each test input [19]. Here we introduce two state-of-the-art approaches: **DeepGini** and **ATS**.

DeepGini [12] exploits the gini coefficient for measuring the purity of test inputs (i.e., the likelihood of correct classification). Given a test input x and its prediction probability $y = \langle y_1, y_2, ..., y_M \rangle$ ($\sum_{i=1}^{M} y_i = 1$), DeepGini selects the most uncertain (minimum purity) data based on the output probabilities by:

$$\underset{x \in X}{argmax} \left(1 - \sum_{i=1}^{M} (y_i)^2 \right) \tag{6}$$

ATS [15] introduces the concept of fault pattern based on unlabeled test input's uncertainty and fault direction. A set of intervals are utilized to describe the fault pattern of a test input (more details refer to [15]). Further, ATS proposes a fitness metric to evaluate the pattern difference between test input x and the selected set S. During the selection, ATS first divides the candidate set into m subsets according to the prediction categories of test inputs. Then, ATS selects test inputs with maximum fitness value in each subset. When all test inputs in the candidate set have the same fitness scores, the test input with high uncertainty would be selected.

3 Motivation

Weiss and Tonella [18] have conducted a replicability study to show that compared with other elaborate TS methods such as neuron-coverage-based, uncertainly-based TS methods work surprisingly well for DNN model testing. Despite these favorable results, existing uncertainty-based TS methods suffer from one or more limitations, including 1) Ignoring noisy data in real-world scenarios, 2) Omitting failure-revealing test inputs and including successful ones, 3) Ignoring the diversity of selected test set. This section describes these limitations and the motivation of our approach in detail.

3.1 Motivation for Considering noisy data in Real-World Scenarios

Previous TS works only consider the candidate sets that contain original test input and their variants generated using data mutation operations. However, in practical scenarios, when collecting data, humans may make mistakes, so noise is usually inevitable in the collected data [24]. To understand the impact of noise on state-of-the-art uncertainty-based TS methods, we conduct a preliminary study.

Data Collection. We train two models, LeNet-5&MNIST and VGG-16&SVHN, using different datasets. To investigate the impact of noise on the TS methods, we construct two candidate sets: one without noise, containing failure-revealing and successful test inputs, and one with noise, which adds an additional 20% of noisy data. Specifically, we randomly select 5,000 successful test inputs from each original test set. Next, we generate image variants using seven data mutation operators, including zoom, shift, brightness, rotation, shearing, blur, and contrast ratio. From these variants, we randomly select 5,000 samples that are misclassified by the tested model. In line with prior work [15], four data pollution methods are utilized to generate noisy data. Such methods involve adding irrelevant data (e.g., incorporating Fashion-MNIST data [36] into the MNIST dataset [37]), generating meaningless synthetic data (randomly assigning pixel values based on the image size), introducing repeated data (randomly selecting and adding existing data to the dataset), and creating crashed data (setting a portion of the image's pixel values to zero). We analyze the performance of the TS methods on these candidate sets, enabling us to quantify the impact of noise on their effectiveness.

TABLE 1
The average failure detection rate (%) for DeepGini and ATS on test set with noise vs. test set without noise.

DNN Dataset	Method	Test set	The Number of Selected Test Inputs					p-value
			250	500	750	1000	2000	
LeNet-5 MNIST	DeepGini	without noise	100.00%	99.40%	99.47%	99.20%	98.50%	$p < 0.05$
		with noise	6.00%	19.20%	29.60%	36.20%	52.70%	
	ATS	without noise	99.60%	99.40%	99.07%	98.80%	97.80%	$p < 0.05$
		with noise	47.60%	54.00%	58.80%	61.00%	66.50%	
VGG-16 SVHN	DeepGini	without noise	99.20%	98.60%	98.00%	98.00%	95.65%	$p < 0.05$
		with noise	53.60%	56.00%	58.40%	60.90%	66.65%	
	ATS	without noise	98.40%	98.20%	97.47%	97.40%	95.30%	$p < 0.05$
		with noise	65.20%	66.60%	68.53%	68.80%	68.25%	

Test Selection Metrics. To answer the above-mentioned questions, we adopt two uncertainly-based selection approaches: DeepGini [12] and ATS [15].

We investigate the impact of noise by applying DeepGini and ATS to candidate sets with and without noise, respectively. We collect n test inputs, selected by TS methods, and obtain their corresponding failure detection performance using Equation 10, where $n = \{250, 500, 750, 1000, 2000\}$. From Table 1, we can observe that when the candidate set contains noise, the performance of the uncertainty-based TS method drops dramatically, regardless of the datasets and DNN models. For example, in LeNet-5&MNIST, DeepGini and ATS exhibit performance gaps of 94.00% and 52.00%, respectively. We utilize the Wilcoxon signed-rank test [38] to verify whether there are statistically significant performance differences for TS methods on the candidate sets. Statistical analysis shows that such performance difference is significant, with a p-value of less than 0.05, regardless of the number of test inputs selected. Building upon our preliminary findings, our proposed approach aims to detect and eliminate noisy test inputs prior to the selection process.

3.2 Motivation for Considering More Than Test Input Uncertainty

Figure 2 visualizes the classification results obtained by a two-class classifier on unlabeled test inputs. The left region represents inputs predicted as the number 7, while the right region represents inputs predicted as the number 1. Intuitively, test inputs closer to the decision boundary are more likely to be misclassified. Consequently, numerous uncertainty indicators [12], [15], [16] have been proposed to identify test inputs near the boundary. However, near-boundary test inputs are not necessarily failure-revealing test inputs. In Figure 2, some test inputs lie near the decision

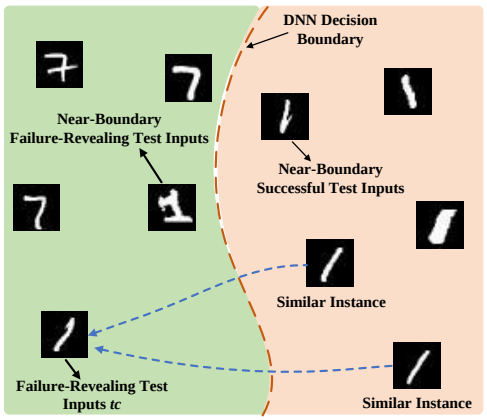


Fig. 2. A motivational example.

boundary, but the DNN model successfully classifies them. What is worse, due to limited model capacity or insufficient training data, DNN models may produce high confidence for failure-revealing test inputs that are far away from the decision boundary (as shown in Figure 2). Yet, existing uncertainty-based TS methods are difficult to select these test inputs.

The above-mentioned cases are because uncertainty-based TS methods only consider the test inputs' uncertainty and ignore the relationship between them. For example, as shown in Figure 2, the failure-revealing test input tc, despite exhibiting high classification confidence, should be identified as suspicious, since similar instances resembling tc are classified as the number 1. This inspires us to leverage the class label information derived from similar instances to identify suspicious test inputs. To achieve this, we design a new identification method to distinguish suspicious and successful test inputs, which contains the following steps: 1) employing a retrieval technique to search for similar instances of each test input (elaborated in Section 4.4.1); and 2) utilizing a voting mechanism to determine the degree of suspiciousness of the test input (outlined in Section 4.4.2).

3.3 Motivation for Considering the Diversity of Test Set

A diverse test set can effectively expose various types of faults in the model under test. Conversely, if the test set used for retraining lacks diversity, it can result in biased training results and reduce the model's accuracy. Although some uncertainty-based TS methods have demonstrated success in selecting failure-revealing test inputs (as shown in Table 1), such methods often overlook the importance of test set diversity. Diversity, particularly input diversity, for test sets has been extensively studied in different forms [39], [40], but in this paper, we emphasize ***the use of test outputs*** as a means of achieving diversity. Essentially, the diversity of output probabilities provides insight into how the model responds to different test inputs. When the model shows similar output probabilities across a range of test inputs, it suggests a perceived similarity among those inputs. In contrast, divergent output probabilities underscore the model's differential responses to diverse inputs, highlighting their distinctions within the feature space. To illustrate, let's consider a candidate test set $X = \{x_1, x_2, x_3, x_4\}$ (refer to the Table 2 for the corresponding prediction probabilities) and a selection size of $n = 2$. DeepGini, according to Equation 6, would select $\{x_1, x_2\}$. However, we prefer to select $\{x_1, x_3\}$ or $\{x_2, x_3\}$. The rationale behind this preference lies in the similarity of output probabilities between x_1 and x_2. In fact, they both exhibit the same failure behavior, where the ground truth is 3 while the predicted class is 1. On the other hand, x_3 demonstrates different output probabilities compared to x_1 and x_2, highlighting a distinct failure scenario where the ground truth is 1 and the predicted class is 2.

TABLE 2
An example of Motivation 3

Candidate Test Set	Output of DNN	DeepGini Score	Prediction Label	Ground Truth
x_1	$\langle 0.55, 0.29, 0.16 \rangle$	0.5878	1	3
x_2	$\langle 0.56, 0.31, 0.13 \rangle$	0.5734	1	3
x_3	$\langle 0.18, 0.59, 0.23 \rangle$	0.5666	2	1
x_4	$\langle 0.02, 0.88, 0.10 \rangle$	0.2152	2	2

In the studies of Alshahwan and Harman [41], [42], we identify evidence supporting the correlation between test output diversity and failure detection. They posit that faults frequently manifest in its observable output, subsequently influencing user decisions. They highlight that the complexity and richness of the output increase the likelihood of faults being exposed. Similarly, Menendez et al. [43] find that enhancing test output diversity augments test set diversity, thereby refining system testing. Drawing from these insights, we propose a novel criterion, based on the uniqueness of the model's output to complement existing test criteria. We expect that raising the diversity of the output could lead to test inputs that are more effective at exposing faults. During each iteration, RTS selects the test inputs that are different from each other in output probability while combining the uncertainty of test inputs.

4 Approach

4.1 Testing Scenario

This paper specifically focuses on computer vision classification tasks, considering that DNNs achieve impressive performance in processing images. In line with previous research on DNN test selection [9]–[12], [15], our study primarily centers around images that possess a clear and unambiguous single class label. It is important to note that our research is confined to the aforementioned scope, and we do not address classification scenarios where images may have multiple or ambiguous labels. By defining this research scope, we ensure that our proposed methodologies, experiments, and findings are applicable and relevant within the context of computer vision classification tasks, specifically involving images characterized by well-defined and singular classification labels.

4.2 Overview

We propose an approach called **RTS** that automates the selection of test inputs for DNN models from unfiltered data without labels, thereby significantly reducing the cost of manual review and labeling. The workflow of RTS is illustrated in Figure 3. Overall, RTS consists of three components: 1) ***Component 1***: dividing noise and non-noise test sets; 2) ***Component 2***: dividing success and non-success test sets; 3) ***Component 3***: prioritizing test inputs of each

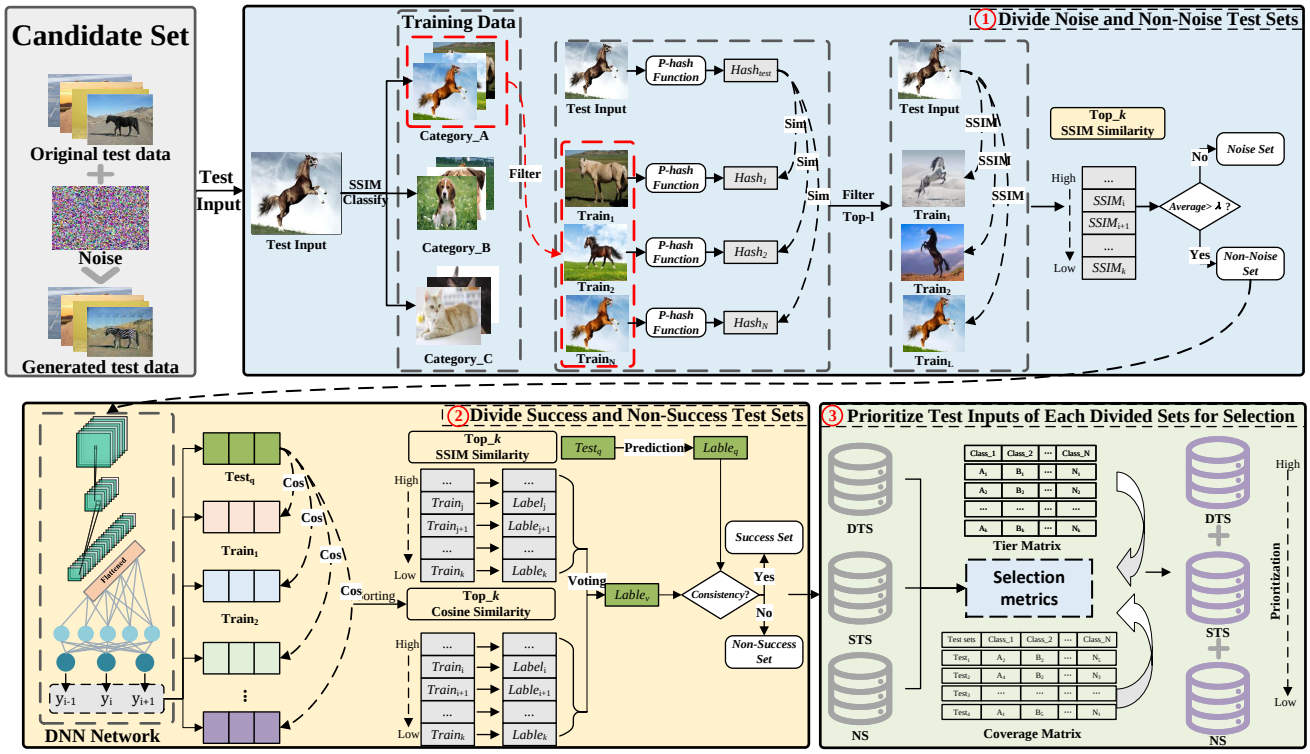
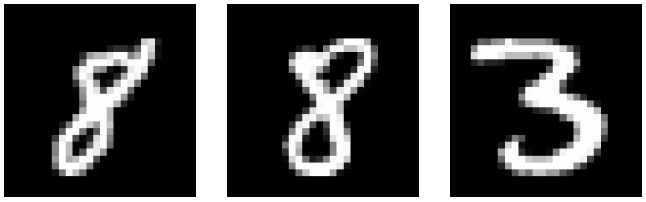


Fig. 3. Overview of **RTS**. Overall, RTS contains three components: 1) ***Component 1***: dividing noise and non-noise test sets; 2) ***Component 2***: dividing success and non-success test sets; 3) ***Component 3***: prioritizing test inputs of each divided sets for selection. ***Component 1*** employs a retrieval technique to search the k most similar training data for each test input. Those test inputs which are not similar to their retrieved neighbors will be regarded as noisy data. ***Component 2*** effectively distinguishes suspicious and successful test inputs by checking the consistency between the estimated labels (generated by majority voting) and predicted labels. ***Component 3*** proposes a novel probability-tier-matrix-based test metric measuring the difference between test inputs in output probabilities to guide test selection in each divided set.

divided sets for selection. Specifically, RTS uses a retrieval technique, structural similarity index (SSIM), and voting mechanism to partition the candidate set (denoted as ***CS***) into noise set (denoted as ***NS***), successful set (in which the successful test inputs located, denoted as ***SCS***), and suspicious set (denoted as ***SPS***), which will be presented in Section 4.3 and Section 4.4. The divided test sets need to satisfy the following properties: (1) $\boldsymbol{CS} = \boldsymbol{SPS} \cup \boldsymbol{SCS} \cup \boldsymbol{NS}$; (2) $\boldsymbol{SPS} \cap \boldsymbol{SCS} = \emptyset \wedge \boldsymbol{SCS} \cap \boldsymbol{NS} = \emptyset \wedge \boldsymbol{SPS} \cap \boldsymbol{NS} = \emptyset$. RTS assigns different selection prioritization to different subsets, i.e., $\text{Order}_{SPS} \gg \text{Order}_{SCS} \gg \text{Order}_{NS}$. This allows RTS to be adaptable to various test selection size. Furthermore, RTS applies a novel probability-tier-matrix-based test metric that quantifies the differences among test inputs within each subset, as explained in Section 4.5. More details about the test case selection process of RTS can be found in Section 4.6.



(a) (b) (c)

Fig. 4. An example on the comparison of SSIM values for different categories of MNIST data.

4.3 Divide Noise and Non-noise Test Sets

Intuitively, compared with valid test inputs, noise data would display more significant differences from the characteristics of the training data. Inspired by this intuition, we retrieve each test input's top-k nearest neighbors among the training set and filter out those test inputs which are not similar to their retrieved neighbors.

Algorithm 1 shows a pseudocode description of Component 1. In the initialization stage (Lines 1 to 3), S is used to store training data randomly selected from the training set under each class, NS is used to store identified noisy data, and Top_s is used to store each test input and its top-k nearest neighbors. Overall, the algorithm can be divided into two parts: *top-k nearest neighbor retrieval* (Line 4-10) and *noise filter* (Line 11-14).

4.3.1 Top-k Nearest Neighbor Retrieval

Similarity Metric. We select the *structural similarity index* [44] (SSIM) as the similarity metric since SSIM can provide acceptable visual agreement and computational overhead. Given two images x_1 and x_2, SSIM compares x_1 and x_2 from three aspects: contrast, localized luminance, and structural similarity, and obtain a scalar score (a value ranging from $[0, 1]$) to quantify the perceptual difference of x_1 and x_2. The higher the value of SSIM, the more similar x_1 and x_2 are. Figure 4 presents an example of the comparison of SSIM values for different classes of MNIST data. Figure 4(a) and Figure 4(b) show different content but share the same label, i.e., the number 8. Figure 4(c) belongs to the number 3. The

Algorithm 1: Component 1 Pseudocode

Input: Candidate set CS
Original training set TS
The number of nearest neighbors k
The number of randomly selected samples n
Threshold λ
Output: Noise set NS
The mapping table storing top-k SSIM nearest neighbors Top_s

1: $S \leftarrow$ **EmptySet()**
2: $NS \leftarrow$ **EmptySet()**
3: $Top_s \leftarrow$ **EmptyMap()**
4: $TS_i\,(i = 1, ..., m) \leftarrow$ **Cluster**(TS)
5: **foreach** TS_i **do**
6: $S_i \leftarrow$ **RandomSample**(TS_i, n)
7: $S \leftarrow$ **Append**(S, S_i)
8: **end for**
9: **foreach** $x \in CS$ **do**
10: $kNNS \leftarrow$ **SearchSsimNeighbors**(x, k, TS, S) // More details in Section 4.3.1
11: $ave \leftarrow$ **AverageSSIM**$(kNNS)$
12: **if** $ave < \lambda$ **then**
13: $NS \leftarrow$ **Append**(NS, x)
14: **end if**
15: $Top_s[x] = kNNS$
16: **end for**
17: Return NS, Top_s

SSIM value between Figure 4(a) and Figure 4(b) is 0.4164, while it is 0.0464 between Figure 4(a) and Figure 4(c).

Retrieval Procedure. One natural way to obtain top-k SSIM nearest neighbors is to compare each test input against all training data and keep the k elements with the highest SSIM values. However, such a way requires large computational overheads, especially when the training set is large. To reduce the computational overheads, it is necessary to reduce the number of training data that need to be processed. One way to do this is to sacrifice the accuracy of the nearest neighbor retrieval by adopting an approximate nearest neighbor retrieval. Therefore, for each test input x in the candidate set ***CS***, RTS follows a three-step process.

***Step 1. Divide Training Set*:** Initially, we partition the original training set into m subsets based on the ground truth (i.e., class labels). These subsets are denoted as $\{TS_1, TS_2, ..., TS_m\}$ (refer to **Line 4** in **Algorithm 1**).

***Step 2. Determine the Most Similar Subset*:** Subsequently, we determine the subset, denoted as TS_h, from the constructed subsets in Step 1 that exhibits the highest similarity to x. To accomplish this, for each subset TS_i (where $1 \leq i \leq m$), we randomly sample n test inputs to construct a sampling set (S_i) representing the TS_i. By performing this step, we generate a series of sampling sets $S = \{S_1, S_2, ..., S_m\}$ (Lines 5-8 in Algorithm 1). The criterion for selecting TS_h is that the corresponding sampling set S_h must satisfy the following condition for all $i \in \{1, 2, ..., m\}$:

$$\frac{1}{|S_h|}\sum_{t \in S_h} SSIM(x, t) \geq \frac{1}{|S_i|}\sum_{t \in S_i} SSIM(x, t) \qquad (7)$$

Above process allows us to focus on computing the similarity specifically between x and TS_h, rather than considering the entire original training set. By narrowing down the scope, we can effectively streamline the computational requirements.

***Step 3. Determine Top-*k *SSIM Nearest Neighbors*:** We utilize *perceptual hashing* [45] (for short *p*-hash) to speed up nearest neighbor retrieval. *P*-hash algorithm generates a fingerprint for each image, allowing visually similar images to be mapped to the same or similar hash codes. Hence, we retain l elements using *p*-hash and the rest of the dissimilar instances are filtered out, where l should be greater than k, and $l = 0.3 \times |TS_h|$ in this paper. While *p*-hash incurs low time overhead, it does not provide a comprehensive assessment of image quality. To ensure accurate nearest neighbor retrieval, we use SSIM to select the k elements among l elements (**Lines 9-10 in Algorithm 1**).

Note that in our approach, the *p*-hash algorithm is initially employed to filter images possessing similar local structures. This is subsequently followed by the application of the SSIM algorithm to the filtered images to derive a more comprehensive global similarity measure. Although the *p*-hash algorithm predominantly underscores local information, it maintains the capacity to retain images exhibiting notable global similarity. This principle is intuitively sensible since, in real-world applications, a collective of similar local structures often suggests a pervasive similarity. The preliminary utilization of *p*-hash to diminish the candidate image set considerably lowers the number of image pairs necessitating global matching, thereby bolstering computational efficiency. Crucially, this minimization in computational demand marginally impacts the capability to identify matches in global information, as corroborated in Table 7.

4.3.2 Noise Filter

For each test input $x \in$ ***CS***, RTS computes the average SSIM value based on retried k nearest neighbors (**Line 11 in Algorithm 1**). If the average value is less than the threshold γ, x is put into ***NS*** (**Lines 12-13 in Algorithm 1**).

The above scheme necessitates the use of a threshold γ to decide the noisy data. A naive approach may be to choose a global threshold. Nevertheless, the semantic similarity of images varies distinctly between classification tasks, rendering a "global threshold" less applicable. Therefore, we have designed an adaptive way of determining γ. Specifically, we randomly sample 10% of data from the training set. Then, following the above process, we retrieve each data's top-k SSIM nearest neighbors, compute the average SSIM values and obtain the minimum element (i.e., $SSIM_{min}$) from all average SSIM values. Since there may be labeling errors in the training corpus [46], [47], we finally set $\gamma = SSIM_{min} + SSIM_{th}$, where $SSIM_{th} = 0.05$ in this paper.

4.4 Divide Success and Non-Success Test Sets

Through Component 1, we obtain a filtered candidate set ***CS'*** containing successful test inputs, some failure-revealing test inputs, and possibly residual noisy data. As discussed in Section 3.2, we propose a new strategy to identify suspicious test inputs using the class labels of similar instances of test inputs. This brings two challenges: 1) determining the similar instances for each test input, and 2) determining the suspiciousness of a test input based on the class labels of its similar instances. To address the first challenge, we retrieve the similar instances of each test input from the training set using two perspectives: original image features and prediction probability. Regarding the second challenge, we employ a voting mechanism to identify suspicious test inputs based on the retrieved instances. In ensemble learning,

majority voting [27] is a straightforward weighting method that selects the class with the highest vote count as the final decision. Next, we provide details about how to separate the ***SCS*** from the ***CS′***.

4.4.1 Similar Instance Retrieval

First, for each test input x, we consider its similar instances (i.e., *voters* in majority voting) based on two perspectives: 1) Type 1: k training data with maximum similarity to x in terms of original image features; 2) Type 2: k training data with maximum similarity to x in terms of prediction probability. The rationale behind type 1 voters is that their visual resemblance to x suggests that x should be assigned the same class label as these voters. For type 2 voters, similar prediction probability implies that the high-dimensional feature vector (created by the DNN model) is similar. Thus, x is likely to have the same ground truth as such vectors. In Component 1, we have obtained each test input's top-k SSIM nearest neighbors according to original image features, i.e., $S = Top_s[x]$. As for the type 2 voters, RTS calculates the cosine similarity between each test input and the training data using prediction probabilities and obtains the top-k cosine nearest neighbors, denoted as C. Ultimately, we combine these two types of voters to create the set $V = S \cup C$.

4.4.2 Filtering Successful Test Inputs

Algorithm 2 provides a pseudocode description of Component 2. In the initialization stage (Lines 1), SCS stores identified successful test inputs. RTS runs the DNN model on the training data to obtain the prediction probabilities Y (Line 2). For each test input $x \in CS'$, RTS captures the prediction probability y and collects its prediction label $\mathcal{L}_p$ based on y (Line 4-5). To construct the type 2 voters (represented as C), RTS calculates the cosine similarity between y and Y and merges C with type 1 voters ($Top_s[x]$, generated by Component 1) to form the set V (Line 6-7). Based on the $V = \{v_1, v_2, ..., v_{2k}\}$, RTS records each element's ground truth (class label) and computes the frequency of labels. By employing majority voting, RTS selects the label with the highest frequency as the estimated label $\mathcal{L}_v$ for x (Line 8). If $\mathcal{L}_p$ is equal to $\mathcal{L}_v$, x is included in the set ***SCS***.

Algorithm 2: Component 2 Pseudocode

Input: Filtered candidate set CS'
Original training set TS
The mapping table storing top-k nearest neighbors Top_s
The number of nearest neighbors k
Output: successful set SCS

```
1: SCS ← EmptySet()
2: Y = Run(DNN, TS)
3: foreach x ∈ CS' do
4:     y = Run(DNN, x)
5:     L_p ← argmax(y) // x's prediction label
6:     C ← SearchCosNeighbors(x, k, Y, y)
       // More details in Section 4.4.1
7:     V = Top_s[x] ∪ C
8:     L_v ← Vote(V)
       // Use the majority voting results as the
       actual label
9:     if L_v == L_p then
10:        SCS ← Append(SCS, x)
11:    end if
12: end for
13: Return SCS
```

4.5 Prioritize Test Inputs of Each Divided Sets for Selection

As described in Section 3.3, we propose a novel test metric to guide the test selection by considering both the uncertainty and diversity of test input. During selection, RTS first finds the candidate inputs that have the maximum difference in output probabilities, compared to the test inputs already selected. Then, the test input with the largest uncertainty would be selected from the candidate inputs. For quantifying uncertainty, we adopt a measure introduced in prior research [15], where the uncertainty of a test input x is defined as $\text{uncertainty}(x) = 1 - \max(y_i)$. The uncertainty measure we employ is based on Max*p* [48], an active learning method. In Max*p*, the instance with the *least confident* prediction is chosen, determined by the maximum prediction probability. Both the uncertainty formula we use and DeepGini can be used to measure the model's confidence in the classification results. However, given that Max*p* consistently demonstrates similar or even better performance than DeepGini (as evident in Tables 5 and 6), we opt to define our uncertainty calculation based on Max*p*'s test selection criterion, rather than directly adopting DeepGini's uncertainty measure. In the following sections, we will describe how we measure the difference between test inputs.

4.5.1 Probability Tier Matrix Construction

First, we discretize the prediction probabilities based on our proposed *probability tier matrix*. Discretizing the output probabilities is beneficial for the following reasons: 1) DNN models generate continuous output probabilities, implying that even a slight variation in the test input can result in different output probabilities. By discretizing the prediction probabilities, we establish distinct tiers or intervals that enable us to capture and differentiate meaningful differences in the output probabilities. 2) As our goal is to select test inputs with diverse output behaviors, comparing and ranking continuous output probabilities directly becomes challenging. By discretizing the prediction probabilities, we can simplify the comparison process, allowing us to identify and prioritize test inputs based on their respective probability tiers.

Specifically, we record the prediction probability of each test input and determine the distribution range for each class. we partition the distribution range of each class into d parts, based on a predefined number of divisions, thus constructing an probability tier matrix: $Matrix(P = \{p_1, p_2, ..., p_m\}, L = \{L_1, L_2, ..., L_m\})$, where P represents the class label set and L corresponds to the level set. Each class label p_i has d levels, i.e., $L_i = \{Ind_1, Ind_2, ..., Ind_d\}$ where Ind_j stores the probability range for the j-th level. Given an output vector y, for i-th class label's prediction probability y_i, we locate the corresponding level set L_i and determine the level index of y_i within L_i. Based on the constructed probability tier matrix, we convert the continuous prediction probabilities into a discrete sequence of level indexes. Note that we assign the lowest level index to the prediction class label.

Example 1. Here, we introduce a 4-label classification example to illustrate the probability tier matrix construction and how to transform the test inputs. As shown in Table 3,

TABLE 3
An example to illustrate probability tier matrix construction.

Test Set	Prediction Probabilities				Transformed Test Set	Level Index			
	1	2	3	4		1	2	3	4
x_1	0.50	0.20	0.20	0.10	x_1	1	1	2	1
x_2	0.12	0.30	0.45	0.13	x_2	1	1	1	1
x_3	0.08	0.16	0.06	0.70	x_3	1	1	1	1
x_4	0.23	0.60	0.05	0.12	x_4	2	1	1	1
x_5	0.50	0.23	0.18	0.09	x_5	1	1	2	1
Distribution Range	$[0,08, 0.50]$	$[0.16, 0.60]$	$[0.05, 0.45]$	$[0.09, 0.70]$	**Tier Matrix**	Ind_1: $[0.08, 0.22)$ Ind_2: $[0.22, 0.36)$ Ind_3: $[0.36, 0.50]$	Ind_1: $[0.16, 0.31)$ Ind_2: $[0.31, 0.46)$ Ind_3: $[0.46, 0.60]$	Ind_1: $[0.05, 0.18)$ Ind_2: $[0.18, 0.31)$ Ind_3: $[0.31, 0.45]$	Ind_1: $[0.02, 0.22)$ Ind_2: $[0.22, 0.42)$ Ind_3: $[0.42, 0.70]$

Suppose that a test set T = {x_1, x_2, x_3} and a DNN model under test with 3 categories.

Test Set (T)	Category				Uncertainty
	1	2	3	4	
x_1	3	1	1	1	0.24
x_2	2	1	2	1	0.23
x_3	1	1	1	2	0.4

Consider RTS, *UncoverCombinations* = {1: {3, 2, 1}, 2: {1}, 3: {2, 1}, 4: {2, 1}}.
Step 1: φ (x_1) = {1: {3}, 2: {1}, 3: {1}, 4: {1}}, Fitness (x_1, S) = | φ (x_1) | = 4
φ (x_2) = {1: {2}, 2: {1}, 3: {2}, 4: {1}}, Fitness (x_2, S) = | φ (x_2) | = 4
φ (x_3) = {1: {1}, 2: {1}, 3: {1}, 4: {2}}, Fitness (x_3, S) = | φ (x_3) | = 4
x_3 is selected as the next test input for *S*, i.e., *S* = < x_3 >, since x_3 has maximum uncertainty under same fitness function values.
Step 2: Update *UncoverCombinations* = {1: {3, 2}, 3: {2}, 4: {1}}.
φ (x_1) = {1: {3}, 2: {1}, 3: {1}, 4: {1}}, Fitness (x_1,S) = | φ (x_1) \ φ (*S*) | = 2
φ (x_2) = {1: {2}, 2: {1}, 3: {2}, 4: {1}}, Fitness (x_2,S) = | φ (x_2) \ φ (*S*) | = 3
x_2 is selected as the next test input for *S*, i.e., *S* = < x_3 , x_2>.

Fig. 5. An illustrative example of selecting tests based on probability tier matrix

for a test set consisting of five test inputs $\{x_1, x_2, x_3, x_4, x_5\}$, we collect each class's probability distribution range $L = \{[0.08, 0.50], [0.16, 0.60], [0.05, 0.45], [0.09, 0.70]\}$ respectively. Assuming the number of divisions is 3, for each class's probability range, such as $[0.08, 0.50]$ of label 1, we compute division length $l = \frac{0.50-0.08}{3} = 0.14$ and construct the corresponding level set $L_1 = \{Ind_1 : [0.08, 0.08 + 0.14), Ind_2 : [0.08 + 0.14, 0.08 + 0.28), Ind_3 : [0.08+0.28, 0.50]\}$. The above-mentioned process is repeated for each class, and we finally obtain the probability tier matrix shown in **Tier Matrix** row. For test input x_1, the prediction probability for class label 3 is 0.2. We locate the level Ind_2: $[0.18, 0.31)$ in which 0.2 falls based on the level set $L_3 = \{Ind_1$: $[0.05, 0.18), Ind_2$: $[0.18, 0.31), Ind_3$: $[0.31, 0.45]\}$, thus converting the prediction probability 0.2 to the discrete level index 2. Since the prediction label of x_1 is class label 1, we directly set the level index of class label 1 to the lowest level. The last four columns of Table 3 give the transformed level index corresponding to the five test inputs.

4.5.2 Select Tests Based on Probability Tier Matrix

With the help of the probability tier matrix, each test input x can be represented by a discrete sequence of level indexes. Based on this, we adopt an additional greedy algorithm to select the test input most different from the selected set S.

Specifically, each test input can cover some *class-level combinations*. In **Example 1**, the level index sequence of x_1 is $\langle 1, 1, 2, 1\rangle$. Thus, the *class-level combinations* covered by x_1 are (1), (1), (2), and (1). To more clearly describe our method, we define a function $\varphi(x)$ that obtains the *class-level combination* covered by x. Further, we define the function $\varphi(T)$ that

Algorithm 3: Component 3 Pseudocode

Input: Test sets to be sorted T
The number of test inputs to be selected SN
Output: Selected test input set S
Combinations ← **`EmptyMap()`**
UncoverCombinations ← $\varphi(T)$
tempCombinations ← *UncoverCombinations*
foreach i $(1 \le i \le |T|)$ **do**
  Combinations[i] ← $\varphi(T[i])$
  `// Calculate each test input's class-level combinations.`
end for
flag ← **false**
while true do
  if *flag* **then**
    UncoverCombinations ← *tempCombinations*
    `// Restart the process.`
  end if
  MaxCombination ← **`EmptySet()`**
  MaxFitness ← 0
  `// Maximum Fitness function value.`
  foreach i $(1 \le i \le |T|)$ **do**
    fitness ← |*UncoverCombinations* ∩ *Combinations[i]* |
    `// Calculate each test input's fitness function value.`
    if *fitness* > *MaxFitness* **then**
      MaxFitness ← *fitness*
      MaxCombination ← **`EmptySet()`**
      MaxCombination ← **`Append`**(*MaxCombination*, $T[i]$)
    end if
    if *fitness* == *MaxFitness* **then**
      MaxCombination ← **`Append`**(*MaxCombination*, $T[i]$)
    end if
  end for
  if *MaxFitness* == 0 **then**
    flag = **true**
  else
    flag = **false**
  end if
  k ← **`MaxUncertainty`**(*MaxCombination*)
  `// Choose the index of test input that have maximum uncertainty`
  $S \leftarrow S \succ \langle T_k \rangle$
  if $|S| == SN$ **then**
    return S
  end if
  UncoverCombinations ← *UncoverCombinations* \ *Combinations[k]*
end while
return S

returns the set of all *class-level combinations* covered by all test inputs in T, i.e.

$$\varphi(T) = \bigcup_{x \in T} \varphi(x) \tag{8}$$

We define a fitness metric to quantify the difference of each test input x against the selected set S, denoted as $\text{Fitness}(x, S)$, i.e.

$$\text{Fitness}(x, S) = |\varphi(x) \setminus \varphi(S)| \tag{9}$$

Algorithm 4: RTS Pseudocode

Input: candidate set CS. Original training set TS.
The number of nearest neighbors k.
The number of of randomly selected samples $n = 50$.
Noise threshold λ.
The number of test inputs to be selected SN.
Output: selected set S

```
1: NS, Tops ← FilterNoise(CS, TS, k, n, λ)
   // Refer to Component 1
2: NonNoise ← {CS \ NS}
3: SCS ← FilterSuccess(NonNoise, TS, Tops, k)
   // Refer to Component 2
4: SPS ← {NonNoise \ SCS}
5: S ← EmptySet()
6: SeletSize ← SN
7: OrderSet ← {SPS, SCS, NS}
8: foreach T ∈ OrderSet do
9:    if SeletSize > |T| then
10:      S ← Append(S, T)
11:      SeletSize ← SeletSize − |T|
12:   else
13:      S' ← SelectTest(T, SeletSize)
          // Refer to Component 3.
14:      Return S
15:   end if
16: end for
```

where $\varphi(x) \setminus \varphi(S) = \{t \in \varphi(x) \mid x \notin \varphi(S)\}$, and $|\cdot|$ returns the length of corresponding set. In other words, the fitness metric would pick the following test input that covers the maximum number of *class-level combinations* not covered by S.

Algorithm 3 shows a pseudocode description of Component 3. In Algorithm 3, *Combination* and *UncoverCombinations* are used to store *class-level combinations* of each test input and test set, respectively (Line 1-5). Then, RTS calculates the fitness function value for each test input and retains the test inputs with the maximum value in *MaxCombination* (Line 14-24). For the elements in *MaxCombination*, RTS selects the test input with the maximum uncertainty as the next test input to be added to S (Line 30-31). Before selecting the next test input, RTS examines whether there are any *class-level combinations* that are not covered by the test input in S: If not, the remaining test inputs are selected by restarting the previous process (Line 9-11). Once a test input is selected, RTS updates the set of uncovered *class-level combinations* (Lines 35). This process is repeated until all the elements of T have been added to S or the size of S is equal to SN (Lines 32-34 and Line 37).

To further explain the details of the test selection, Figure 5 illustrates an example of the selection process. RTS counts the number of *class-level combinations* covered by per test input. Since there are three candidates with the same maximum fitness values, RTS chooses the x_3 with maximum uncertainty and adds it to S. RTS then updates the set *UncoverCombinations*, and calculates the fitness function for each candidate: $\text{Fitness}(x_1, S) = 2$ and $\text{Fitness}(x_2, S) = 3$. Since x_2 has the greater fitness value, it is selected as the next test input and added to S.

4.6 Robust Test Selection Algorithm

Algorithm 4 provides a pseudocode description of RTS. We divide the whole selection procedure into three parts: filtering noisy data, filtering successful test inputs, and test selection. From Line 1 to 3, we divide the raw candidate set (***CS***) into ***NS***, ***SCS***, and ***SPS*** through Component 1 and 2. Then, RTS assigns different selection priorities to different subsets, i.e., $\text{Order}_{SPS} \gg \text{Order}_{SCS} \gg \text{Order}_{NS}$ (Line 6). This means that test inputs from ***SPS*** are prioritized over ***SCS***, which is prioritized over ***NS***. Starting from Line 7, RTS begins the test selection process. It first selects test inputs from the ***SPS*** subset. If the required number of test inputs (SN) is greater than the size of ***SPS***, RTS utilizes Component 3 to select the remaining test inputs from the ***SCS*** subset. This process continues until the required number of test inputs is reached or all subsets are exhausted (Line 7-14). Additionally, based on empirical evidence, the value of k is set to 5.

Note that we do not immediately delete or discard the noisy data identified by the RTS. This decision is based on two key considerations: 1) ***Precision of Noisy Data Identification***: Although we want to precisely identify all the noisy data in the candidate test set, it is a non-trivial task, and RTS may misclassify some valid test inputs as noise as well. Removing ***NS*** set directly would lose these valid test inputs. 2) ***Availability of Labeling Resources***: When testers have sufficient labeling resource, they can further examine the data in the ***NS*** set to select an adequate number of test inputs.

TABLE 4
Dataset and DNN models.

Dataset	Description	DNN Model	#Parameters	Layers	$\text{Accuracy}_{\text{raw}}$	$\text{Accuracy}_{\text{new}}$
MNIST	28 × 28 hand-writtern digits	LeNet-1	7,206	5	98.78%	89.31%
		LeNet-5	44,404	7	99.03%	90.18%
CIFAR-10	32 × 32 color images	ResNet-20	273,066	20	75.61%	65.08%
		VGG-16	15,769,930	21	86.71%	79.16%
Fashion	28 × 28 gray-scale images	LeNet-1	7,206	5	89.25%	77.16%
		ResNet-20	272,778	20	90.48%	79.14%
SVHN	street view numbers	LeNet-5	61,984	7	88.51%	77.51%
		VGG-16	15,769,930	21	95.37%	89.11%
CIFAR-100	32 × 32 color images	ResNet-20	272,778	20	51.46%	47.65%
		ResNet-32	470,868	32	52.77%	47.96%
Caltech-101	200 × 200 color images	ResNet-20	590,629	20	72.13%	65.51%
		ResNet-32	786,405	32	71.52%	64.86%

1. $\text{Accuracy}_{\text{raw}}$ refers to the accuracy of model on **original test set**.
2. $\text{Accuracy}_{\text{new}}$ refers to the accuracy of model on our **newly constructed test set**.

5 Experiments Design

This section details our experimental settings, including the data set, the DNN model under test in the experiment, candidate set construction, and a series of research questions. Thanks to the contributions of Gao et al. [15], we implement our algorithm based on their testing framework upon Keras 2.3.1 with TensorFlow 1.13.1. We release these data and our scripts for follow-up work[2]. All experiments are conducted on a Ubuntu 20.04 with four NVIDIA GeForce RTX 3090 GPUs, one 12-core processor, and 256GB memory.

5.1 Datasets and DNN models

Following previous research [12], [15], we adopt four well-known publicly available DNN datasets: MNIST, Fashion-MNIST, CIFAR-10, and SVHN. To further demonstrate the generalizability of RTS, we additionally add two more complex datasets: CIRAR-100 and Caltech-101. Table 4 presents detailed information on these datasets. We measure model performance in terms of accuracy, the original metric

2. https://github.com/swf1996120/RTS

employed for the tasks and datasets under investigation. Since we do not use all the training data and do not use data augmentation techniques to train the model, the accuracy achieved by our models (as shown in $\textbf{Accuracy}_{\textbf{raw}}$ column of Table 4) may not be comparable to state-of-the-art results reported in the literature. Nevertheless, our focus is not on achieving the highest accuracy but rather on investigating the effectiveness of TS methods in detecting failures of the pre-trained DNNs. MNIST [37] is a grayscale image dataset of hand-written digits with ten labels, e.g., 1, 2. Fashion-MNIST [36] includes ten categories of fashion products, e.g., coats and shirts. Each class consists of a training dataset of 6,000 images and a test dataset of 1,000 images. Each input data is a 28×28 greyscale image. CIFAR-10 [49], [50] dataset (abbreviated as CIFAR) contains 60,000 32×32 images with three channels, e.g., airplane, bird, from which 50,000 are training data, and 10,000 are test data. SVHN [51] is a real-world image dataset obtained from house numbers in Google Street View images for developing machine learning and object recognition algorithms. CIFAR-100 [49] is a well-known benchmark dataset extensively utilized in computer vision research. It comprises 100 classes, which are organized into two levels of granularity: coarse and fine. The coarse-level classification consists of 20 main classes that represent high-level concepts, such as aquatic mammals, insects, and trees. Our paper primarily focuses on the challenging task of coarse-level classification due to the complexities involved in model training. Caltech-101 [52] consists of approximately 9,000 images categorized into 101 classes, with each class containing roughly 40 to 800 images. Additionally, we consider five famous DNN models with different scales to ensure the generalizability of our experiments, which are LeNet-1, LeNet-5 [53], VGG-16 [54], ResNet-20 [55], ResNet-32 [55]. To ensure the evaluation result is stable, we train two different DNN models separately for each dataset.

5.2 Candidate Test Input Collection

In line with prior work [15], for candidate test input construction, we generate additional new test inputs using seven well-used benign perturbations for each dataset, while preserving their original label. Such benign data augmentation operators involve zoom, shift, brightness modification, rotation, shearing, blur, and contrast ratio adjustment [20], [56]. Image transformations apply basic geometric transformations to simulate different real-world conditions, which can reflect realistic data mutations in the usage scenario. In this study, we adhere to the image transformation procedures specified in previous literature [10], [15], [57]. Furthermore, we conduct careful inspections of the transformed images to ensure their semantic consistency with the original ones.

We apply all the above-mentioned data augmentation operators to each original test input so that each augmentation operator produces the same amount of data. Subsequently, we evenly partition the original test set and each generation set into two parts: one for constructing the candidate test set and the other for constructing the new test set. For example, for the Fashion-MNIST (whose test set size is 10,000), a candidate set and a new test set with the same size of 40,000 is constructed, respectively, including 5,000 original test data and 5,000 $\times$ 7 generated test data for seven augmentation operators. Note that TS methods can only select test inputs from the constructed candidate test set, and the new test set is only used to evaluate the DNN's performance. To simulate various types of invalid data that could exist in an unfiltered dataset, we follow prior work [15] and construct four polluted candidate sets. Each polluted set corresponds to a specific data pollution method and contains 20% invalid data, where the percentage and amount of noise are exactly as set by Gao et al [15]. Such methods involve adding irrelevant data (e.g., incorporating Fashion-MNIST data [36] into the MNIST dataset [37]), generating meaningless synthetic data (randomly assigning pixel values based on the image size), introducing repeated data (randomly selecting and adding existing data to the dataset), and creating crashed data (setting a portion of the image's pixel values to zero). For distinguishing noisy data from valid test inputs, except for repeated data which retains the original class labels, the labels of the other types of noisy data are set to "-1".

In summary, for each dataset, we construct a new test set and five candidate sets, including a purely valid candidate set and four candidate sets containing partially invalid data. In the subsequent experiments, we apply a specific TS method to each candidate set to select test inputs. These inputs are then used to evaluate the failure detection rate (**RQ1**) of the TS methods and the repair/optimization performance (**RQ2**). Consistent with previous study [15], we report the average results across all candidate sets as the final performance of the TS method. This way enables us to comprehensively evaluate the performance of TS methods across different testing scenarios. Note that, the original test set may have some limitations when it comes to accurately evaluating the robustness of a model, particularly in terms of capturing specific data transformation modalities. Hence, we utilize the newly constructed test set to assess whether the selected inputs can further optimize the model's performance. As indicated in the $\textbf{Accuracy}_{\textbf{new}}$ column of Table 4, the performance of the tested models degrades to some extent on the new test set.

5.3 Baseline Approaches

We conduct extensive experiments to evaluate the effectiveness of RTS and compare it with 14 baseline approaches, which can be categorized into four types: neuron-coverage-based, uncertainty-based, active learning, and other TS methods based on different concepts.

1) **Neuron-Coverage-Based TS Method.** We compare RTS with four well-known neuron coverage criteria (i.e., **NBC**, **TKNC**, **SNAC** [10] and **NAC** [9]). Furthermore, we follow the default settings of the original papers to specify the configurable parameters of the neuron coverage criteria.

2) **Uncertainty-Based TS Method.** We chose **DeepGini** [12], **ATS** [15], and **variance score** (VS) [58] as representatives of the uncertainty-based TS method. Please refer to Section 2.4 for details of DeepGini and ATS. **VS** estimates the uncertainty by sampling k dropped-out models and computing the variance of their resulting prediction probabilities over a test input.

3) **Active learning Method.** Active learning strategically assesses the informativeness of unlabeled instances to bolster model training. In this context, we employ three

representative active learning methods: **Max***p* [48], **Entropy Sampling with Dropout** (EDrop) [59], and **CLUE Sampling** (CLUE) [60]. **Max***p* selects the instance with the *least confident* prediction, based on the maximum prediction probability. **EDrop** serves as a fundamental active learning strategy, selectively sampling instances that embody maximal model uncertainty, quantified through entropy. Conversely, **CLUE** adopts an uncertainty-weighted clustering approach, pinpointing instances for labeling that not only exude model uncertainty but also maintain diversity in the feature space, thereby ensuring a breadth of information is incorporated during model refinement.

4) **Other TS methods.** We select three widely-used TS methods based on different conceptions, including random sampling (for short, ***RS***), **CES**, **LSA**. ***RS*** may be the simplest test selection technique, which utilizes a random manner to select test input from the candidate set according to the target size. **CES** [61] is a test selection method based on conditioning, which aims to select a small subset from the original test inputs to estimate the performance of the target DNN model via cross-entropy. **LSA** (**L**ikelihood-based **S**urprise **A**dequacy) [16] measures how *surprising* or out-of-distribution the DNN activations for a given test input are with respect to the activations observed on the training set. The higher the LSA value of the test input, the more surprising it is for the DNN. We introduce a variant of DeepGini, denoted as **DMsp**, serving as one baseline method, which implements a pre-filtering step for out-of-distribution (OOD) data prior to utilizing DeepGini for test input selection. The **DMsp** employs the Maximum Softmax Probability [62] to discriminate between normal and abnormal test inputs, establishing itself as a widely accepted baseline. In this work, the minimum-maximum softmax probability value derived from the training data is employed as a threshold to discern normal samples from their abnormal counterparts.

5.4 Research Questions

We expect the test inputs selected by RTS can reveal more failures of tested model and fix/optimize the tested model's incorrect behaviors. Thus, it is necessary to evaluate the effectiveness of RTS in terms of two aspects: failure detection performance and model optimization capability. Furthermore, we further explore the contribution of each component of RTS. The empirical study is meant to answer the following three research questions (RQ):

- **RQ1:** *To what extent can RTS outperform baseline approaches in detecting failures?*
- **RQ2:** *Does retraining with RTS-selected test inputs result in more effective model?*
- **RQ3:** *How effective is each component of RTS?*
  - **RQ3.1:** *Do all components of RTS contribute to the effectiveness of failure detection?*
  - **RQ3.2:** *Can RTS's retrieval strategies reduce time overhead effectively?*

5.4.1 RQ1: Failure Detection

For **RQ1**, we adopt the failure detection rate (denoted as FD_{ratio}), which is defined as the proportion of failure-revealing test inputs in the selected set S. Given a selected set S, FD_{ratio} is computed as follows:

$$\mathrm{FD}_{ratio}(S) = \frac{|S_{wrong}|}{|S|} \tag{10}$$

where $|S_{wrong}|$ is the number of misclassified test inputs, and $|S|$ is the size of the selected set. We select the 10% size of each candidate set for each DNN&Dataset combination and collect the corresponding FD_{ratio}. Higher FD_{ratio} values indicate better performance in deteting defects.

5.4.2 RQ2: Optimization Effectiveness

As mentioned in Section 1, developers can fix incorrect/unexpected behaviors of DNNs by retraining them with new data and additional training epochs. Therefore, **RQ2** evaluates the effectiveness of TS methods in fixing or optimizing DNN models. We use the new test set $Test_{new}$ constructed in Section 5.2 to evaluate the performance of the tested model, obtaining accuracy results denoted as Auc_{bef} (see the **Accuracy$_{\text{new}}$** column of Table 4). We then retrain the DNN model by combining test inputs of different sizes (2.5%, 5%, 7.5%, 10%) selected by TS methods with the original training set. We evaluate the retrained model on $Test_{new}$, obtaining accuracy results denoted as Auc_{after}. The optimization effectiveness (**OE**) is defined as:

$$\mathrm{OE} = \mathrm{Auc}_{after} - \mathrm{Auc}_{bef} \tag{11}$$

We perform the experiment three times for all combinations to reduce the impact of random errors in the retraining process. Furthermore, we use the Wilcoxon rank-sum test [63] to determine whether RTS significantly outperforms baseline approaches at the 95% significance level.

5.4.3 RQ3: Ablation Experiment

We conduct a series of ablation experiments to thoroughly investigate the effectiveness and efficiency of RTS.

RTS demonstrates its effectiveness in enhancing the failure detection performance of TS, particularly in complex testing scenarios. This improvement is attributed to three crucial components: a noise filtering component, a voting mechanism for identifying failure-revealing test inputs, and an innovative test metric that quantifies test case differences better. In **RQ3.1**, we investigate the impact of these components on RTS's effectiveness by comparing it to three variants: 1) **RTS-1**, which excludes the noise filtering component that divides the ***NS*** set; 2) **RTS-2**, which excludes the voting mechanism that divides the ***SCS*** set; and 3) **RTS-3**, which uses a random test selection method instead of component 3. Since retraining the model is time-consuming, we only compare the FD_{ratio} of RTS and its three variants.

To expedite the nearest neighbor retrieval process, we implement two key steps (refer to Section 4.3.1): 1) retrieving the subset of the most similar instances for each test input based on the sampled training data; and 2) speeding up the search for the *k* nearest neighbors using the *p*-hash algorithm. In **RQ3.2**, we investigate the impact of these steps on the efficiency of retrieving the top-k nearest neighbors in RTS. To assess this effect, we compare RTS against two variants: 1) **RTS-OBO**, which sequentially compares training data to identify the most similar instances; and 2) **RTS-*p*hash**, which

TABLE 5
The average failure detection rate (%) for each configuration. Values highlighted in red and blue indicate the best and second best.

Selection Rate	Dataset	Model	RTS	Neuron-Coverage-Based				Uncertainty-Based			Active Learning			Other			
				NAC	NBC	SNAC	TKNC	ATS	Gini	VS	Maxp	EDrop	CLUE	CES	LSA	DMsp	RS
2.5%	MNIST	LeNet5	53.60	10.50	8.88	8.92	10.94	45.12	25.80	42.34	26.82	25.18	22.04	10.20	37.16	26.78	8.64
		LeNet1	55.06	11.06	11.28	9.48	9.94	45.54	30.76	46.56	31.34	31.44	24.16	8.70	38.52	32.86	9.48
	Fashion	LeNet1	60.00	21.18	19.58	19.72	21.98	59.90	57.88	39.34	59.70	52.62	26.92	17.50	30.38	57.90	20.96
		ResNet20	58.08	20.00	19.66	17.54	20.34	55.98	56.18	32.98	55.94	51.00	22.58	19.34	33.74	53.78	19.56
	CIFAR	VGG16	60.26	17.74	16.28	15.62	23.44	57.90	51.84	47.22	50.32	50.78	34.72	23.48	25.70	51.62	19.06
		ResNet20	68.66	31.86	28.78	30.14	30.34	64.62	62.46	47.46	61.86	61.04	43.04	32.06	28.06	60.8	29.88
	SVHN	LeNet5	67.44	19.29	20.98	19.90	20.05	61.45	63.40	39.00	61.83	58.46	34.65	19.98	30.50	63.39	20.18
		VGG16	56.07	9.87	11.18	10.12	12.23	51.93	38.06	41.31	38.14	38.63	18.35	17.33	14.78	38.01	9.64
	CIFAR-100	ResNet20	67.70	45.56	43.92	46.14	46.72	61.32	69.12	53.70	70.98	68.12	54.86	46.90	39.66	69.38	47.68
		ResNet32	69.20	46.00	43.40	46.46	47.20	60.26	70.62	56.72	71.42	70.00	55.82	48.26	43.58	70.54	49.78
	Caltech-101	ResNet20	74.57	28.03	31.48	24.44	40.62	63.70	61.73	69.75	63.58	59.51	52.35	32.72	43.95	69.51	32.22
		ResNet32	72.22	24.69	31.98	27.16	38.89	69.75	60.62	62.47	62.96	64.07	44.07	33.58	42.47	64.32	35.93
	Avg.	——	63.57	23.81	23.95	22.97	26.89	58.12	54.04	48.24	54.57	52.57	36.13	25.84	34.04	54.91	25.25
	Statistical Results	*p*-value	——	***	***	***	***	***	**	***	**	**	***	***	***	**	***
		cliff's delta	——	1.00	1.00	1.00	1.00	0.33	0.32	0.72	0.31	0.47	0.96	1.00	1.00	0.28	1.00
		Effect-size	——	*L*	*L*	*L*	*L*	*S*	*S*	*L*	*S*	*M*	*L*	*L*	*L*	*S*	*L*
5%	MNIST	LeNet5	49.13	9.64	9.27	8.25	9.90	46.47	24.97	41.52	26.17	23.74	21.05	10.25	39.52	26.34	8.68
		LeNet1	50.73	10.37	10.73	9.09	10.30	45.84	30.56	43.25	31.67	31.47	23.22	8.10	34.80	32.45	9.46
	Fashion	LeNet1	57.21	21.05	20.02	19.23	21.42	55.11	57.92	39.71	58.02	52.41	27.19	17.53	30.05	57.91	20.67
		ResNet20	56.42	19.81	18.97	17.62	20.29	53.43	53.78	34.02	53.70	49.20	22.78	19.38	30.64	52.34	19.07
	CIFAR	VGG16	57.02	17.66	16.85	17.09	21.86	55.02	47.98	47.41	48.89	48.43	34.73	24.28	24.23	47.94	18.76
		ResNet20	67.10	31.79	29.96	30.09	31.25	61.96	59.91	48.27	60.32	58.21	42.92	31.91	26.10	58.60	29.92
	SVHN	LeNet5	65.79	20.30	20.55	19.99	20.24	58.49	59.21	38.05	59.72	54.81	34.96	19.57	28.89	59.17	20.13
		VGG16	51.26	9.82	10.02	10.02	10.83	47.95	36.66	40.11	37.05	35.81	18.00	17.10	12.41	36.63	9.62
	CIFAR-100	ResNet20	67.07	46.59	45.47	46.21	46.21	60.51	69.44	53.74	70.53	68.02	54.83	47.50	37.38	69.31	47.35
		ResNet32	67.96	47.07	45.80	47.25	47.47	58.34	69.95	56.39	70.28	69.25	55.73	47.71	39.54	69.95	49.15
	Caltech-101	ResNet20	75.68	30.68	30.43	28.89	34.63	62.22	61.98	67.90	64.07	61.48	49.32	33.09	43.77	68.95	32.72
		ResNet32	72.35	29.01	29.07	30.49	33.21	64.69	59.82	63.09	61.05	64.44	44.26	35.00	44.07	64.01	34.32
	Avg.	——	61.48	24.48	23.93	23.69	25.63	55.84	52.68	47.79	53.46	51.44	35.75	25.95	32.62	53.63	24.99
	Statistical Results	*p*-value	——	***	***	***	***	***	**	***	**	**	***	***	***	**	***
		cliff's delta	——	1.00	1.00	1.00	1.00	0.38	0.26	0.71	0.26	0.38	0.90	1.00	1.00	0.21	0.99
		Effect-size	——	*L*	*L*	*L*	*L*	*M*	*S*	*L*	*S*	*M*	*L*	*L*	*L*	*S*	*L*
7.5%	MNIST	LeNet5	44.45	8.93	9.39	8.22	9.61	44.51	24.71	40.55	25.96	23.23	20.64	10.41	39.23	26.51	8.59
		LeNet1	47.81	10.17	10.68	9.29	10.01	44.18	30.64	40.86	31.64	31.17	22.91	8.28	32.77	32.03	9.30
	Fashion	LeNet1	55.13	20.39	19.84	19.84	20.77	52.76	56.76	39.36	56.91	52.05	27.45	17.36	31.05	56.77	20.61
		ResNet20	54.93	19.63	19.03	18.31	19.47	51.92	52.05	34.49	52.27	48.23	23.23	19.82	31.47	50.65	18.98
	CIFAR	VGG16	55.13	18.12	17.69	17.44	21.40	53.18	46.41	47.76	47.27	46.63	34.54	24.54	23.31	46.35	18.75
		ResNet20	65.07	31.97	30.73	30.63	31.33	60.26	58.71	47.77	59.21	57.27	42.87	31.56	25.49	57.52	30.08
	SVHN	LeNet5	64.15	20.17	20.42	19.97	20.22	56.80	56.93	37.58	57.46	53.23	35.98	19.36	27.85	56.91	19.85
		VGG16	47.65	9.65	9.77	9.90	10.18	44.85	35.59	39.21	36.55	35.05	18.21	17.12	11.20	35.58	9.57
	CIFAR-100	ResNet20	66.54	46.70	46.12	46.77	46.68	59.75	69.47	54.16	70.25	67.71	54.40	47.17	36.08	69.36	47.48
		ResNet32	67.68	47.06	46.47	48.04	48.17	57.79	69.27	56.83	69.75	69.05	55.51	47.63	38.07	69.33	48.84
	Caltech-101	ResNet20	74.36	29.42	29.51	28.03	33.66	61.36	62.26	66.26	63.58	62.43	47.86	32.26	43.42	68.03	30.95
		ResNet32	73.05	28.56	29.79	30.29	32.76	63.83	59.75	62.26	61.28	64.53	43.66	33.79	43.13	63.33	33.46
	Avg.	——	59.66	24.23	24.12	23.89	25.36	54.27	51.88	47.26	52.68	50.88	35.61	25.77	31.92	52.70	24.70
	Statistical Results	*p*-value	——	***	***	***	***	***	**	***	**	**	***	**	***	**	***
		cliff's delta	——	0.97	0.97	0.94	0.94	0.35	0.25	0.63	0.25	0.32	0.83	0.97	1.00	0.19	0.94
		Effect-size	——	*L*	*L*	*L*	*L*	*M*	*S*	*L*	*S*	*S*	*L*	*L*	*L*	*S*	*L*
10%	MNIST	LeNet5	39.84	8.76	9.04	8.40	9.25	42.15	24.09	38.70	25.55	22.92	20.54	10.28	38.62	26.63	8.78
		LeNet1	44.39	9.80	10.26	9.14	9.62	42.22	30.30	38.89	31.52	30.46	23.35	8.15	31.96	31.50	9.31
	Fashion	LeNet1	53.77	20.72	20.06	19.91	20.54	50.87	55.11	39.55	54.90	51.38	27.45	17.50	31.71	55.08	20.60
		ResNet20	54.33	19.76	19.13	18.14	19.35	50.58	50.84	34.60	51.13	46.87	23.63	19.72	31.70	49.79	18.87
	CIFAR	VGG16	53.13	18.39	17.82	17.72	20.57	51.39	45.61	47.05	46.03	45.28	34.41	24.31	22.19	45.63	18.74
		ResNet20	64.42	32.14	30.24	30.64	30.88	58.42	58.19	48.17	57.69	56.52	43.24	31.20	25.18	57.05	30.47
	SVHN	LeNet5	62.59	20.32	20.34	19.83	20.17	55.08	54.74	37.24	55.48	51.79	36.06	19.27	27.47	54.74	19.84
		VGG16	44.33	9.67	9.86	9.75	9.84	41.72	35.63	38.10	36.18	34.35	18.40	16.84	10.68	35.60	9.67
	CIFAR-100	ResNet20	66.84	47.41	46.64	46.68	47.12	58.80	69.38	54.37	69.76	67.61	54.26	46.67	35.49	69.25	47.48
		ResNet32	67.43	47.68	46.71	47.99	48.28	57.40	68.99	57.13	69.22	68.83	55.15	47.68	37.48	68.91	48.33
	Caltech-101	ResNet20	71.20	29.20	29.66	27.65	32.62	61.67	62.10	66.02	62.90	62.84	46.91	32.93	42.84	67.19	30.31
		ResNet32	70.53	28.52	29.75	28.95	32.69	64.23	60.68	62.59	61.48	64.07	43.55	33.49	41.54	63.89	32.38
	Avg.	——	57.73	24.36	24.12	23.73	25.08	52.87	51.30	46.87	51.82	50.24	35.58	25.67	31.40	52.10	24.56
	Statistical Results	*p*-value	——	***	***	***	***	**	**	***	**	**	***	***	***	**	***
		cliff's delta	——	0.92	0.92	0.92	0.92	0.32	0.22	0.55	0.21	0.28	0.78	0.92	0.97	0.17	0.92
		Effect-size	——	*L*	*L*	*L*	*L*	*S*	*S*	*L*	*S*	*S*	*L*	*L*	*L*	*S*	*L*

[1] *** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$, – $p > 0.05$.
[2] *L*, *M*, *S* and *N* represent Large, Medium, Small and Negligible effect size according to Cliff's delta.

eschews the use of the *p*-hash algorithm during retrieval. The efficiency of top-k nearest neighbor retrieval is appraised by timing the instance retrieval process. Besides, we delineate the accuracy of RTS and its two variants in pinpointing noise data, aiming to substantiate whether this approximated nearest-neighbor retrieval mechanism furnishes performance that is commensurate with precise nearest-neighbor retrieval.

6 Result Analysis

6.1 Answer to RQ1: Failure Detection

In particular, since labeling test output is costly and requires large human resources, it would be better if more failure-revealing test inputs could be identified while labeling fewer test inputs. Therefore, we display four selection ratio results, including 2.5%, 5%, 7.5%, and 10%. We compare the average failure detection rate between RTS and baseline methods in Table 5. We use red color to highlight the best FD_{ratio} and blue color to highlight the second best FD_{ratio} for each DNN&Dataset configuration. Table 5 reveals that all neuron-coverage-based TS methods have poor performance in detecting failures. Such observation can be explained as: previous researches [12] has demonstrated that achieving the maximum neuron coverage is an easy task (only a few test inputs) and that neuron coverage is no longer increased by selecting more test inputs. Compared with the RS baseline, both RTS and most uncertainty-based TS methods have a better failure detection capability. Among the 48 cases (6 subjects × 4 selection rates × 2 models), RTS outperforms others in 72.92% of cases (35 out of 48),

TABLE 6
The DNNs' accuracy improvement (%) value after retraining with the selected tests. Values highlighted in red and blue indicate the best and second best.

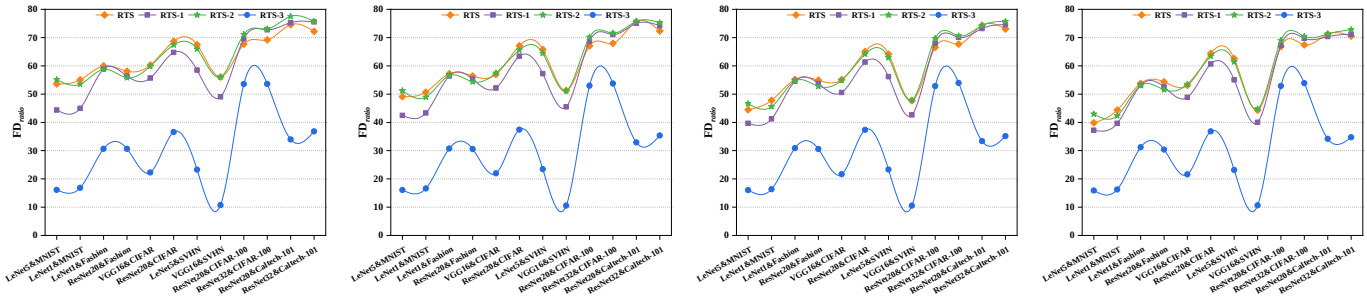
	Dataset	Model	RTS	Neuron-Coverage-Based				Uncertainty-Based			Active Learning			Other			
				NAC	NBC	SNAC	TKNC	ATS	Gini	VS	Maxp	EDrop	CLUE	CES	LSA	DMsp	RS
Select 2.5%	MNIST	LeNet5	3.461	1.908∗	1.901∗	1.979∗	2.309∗	3.084∗	2.125∗	3.491–	2.194∗	2.035∗	2.889∗	2.065∗	3.087∗	2.146∗	2.000∗
		LeNet1	3.288	1.940∗	1.937∗	1.879∗	1.942∗	3.242-	2.187∗	2.608∗	2.351∗	2.339∗	2.648∗	1.512∗	2.901∗	2.563∗	1.982∗
	Fashion	LeNet1	2.501	1.597∗	1.364∗	1.412∗	1.758∗	2.510-	1.965∗	1.923∗	2.078∗	1.924∗	1.840∗	1.357∗	1.630∗	1.906∗	1.647∗
		ResNet20	3.390	0.517∗	0.737∗	1.189∗	1.081∗	3.383-	2.713∗	2.963∗	3.166-	2.804∗	2.970∗	2.314∗	2.038∗	2.728∗	2.165∗
	CIFAR	VGG16	2.500	1.667∗	1.679∗	1.604∗	1.857∗	2.245-	2.148∗	2.014∗	2.207∗	2.293∗	2.115∗	1.908∗	1.610∗	-7.174∗	1.786∗
		ResNet20	1.284	1.073-	1.457-	0.920-	0.696-	1.484-	1.311-	1.011-	1.670-	1.582-	1.318-	1.497-	0.770-	2.152∗	1.197-
	SVHN	LeNet5	1.333	0.375∗	0.384∗	0.403∗	0.343∗	1.182-	0.904∗	0.963∗	0.951∗	0.697∗	1.006∗	0.204∗	0.681∗	0.896∗	0.393∗
		VGG16	3.227	1.439∗	1.570∗	1.589∗	1.694∗	2.878∗	2.275∗	2.862∗	2.366∗	2.082∗	2.170∗	2.010∗	1.106∗	2.234∗	1.615∗
	CIFAR-100	ResNet20	0.917	1.085-	0.824-	0.847-	0.490-	0.573-	1.283-	0.907-	0.675-	0.689-	0.692-	0.530-	0.670-	0.412∗	0.545-
		ResNet32	0.832	0.428-	0.601-	0.321-	0.494-	0.503-	0.570-	0.837-	0.919-	0.751-	0.972-	0.444-	0.757-	0.880-	0.582-
	Caltech-101	ResNet20	2.498	2.050∗	2.384-	2.300-	2.728-	2.883∗	2.204-	2.825∗	2.346-	2.385-	2.982∗	2.044∗	1.557∗	2.431-	2.242-
		ResNet32	2.521	1.419∗	1.018∗	1.621∗	1.764∗	2.521-	2.124∗	2.167∗	2.210∗	2.031∗	2.508-	1.857∗	1.842∗	2.069∗	1.634∗
Select 5%	MNIST	LeNet5	4.456	2.625∗	2.630∗	2.640∗	2.863∗	4.236∗	2.614∗	4.528-	2.982∗	2.642∗	3.786∗	2.761∗	4.172∗	2.934∗	2.657∗
		LeNet1	4.351	2.680∗	2.604∗	2.674∗	2.672∗	4.318-	3.146∗	3.512∗	3.396∗	3.327∗	3.682∗	2.020∗	3.595∗	3.415∗	2.674∗
	Fashion	LeNet1	3.476	2.394∗	2.408∗	2.141∗	2.599∗	3.323∗	2.951∗	2.901∗	3.092∗	2.571∗	2.793∗	2.055∗	2.191∗	2.915∗	2.409∗
		ResNet20	4.465	2.087∗	2.154∗	2.075∗	2.225∗	4.525-	3.782∗	3.841∗	4.185-	4.073-	3.844∗	3.354∗	2.879∗	4.312-	3.428∗
	CIFAR	VGG16	3.289	1.999∗	2.220∗	2.157∗	2.396∗	3.194∗	2.938∗	2.936∗	3.016∗	2.948∗	2.798∗	2.454∗	1.988∗	2.553∗	2.276∗
		ResNet20	2.446	1.408∗	1.628-	1.137∗	1.526∗	2.275-	2.250-	2.010-	2.085-	2.097-	1.845-	1.307∗	1.611-	2.009-	1.956∗
	SVHN	LeNet5	2.738	1.183∗	1.275∗	1.099∗	1.142∗	2.322∗	2.060∗	2.064∗	2.262∗	1.836∗	2.087∗	0.891∗	1.461∗	2.032∗	1.073∗
		VGG16	4.005	2.272∗	2.183∗	2.270∗	2.343∗	3.807∗	3.119∗	3.695∗	3.262∗	2.968∗	2.990∗	2.824∗	1.469∗	3.117∗	2.232∗
	CIFAR-100	ResNet20	1.618	1.273-	1.328-	1.563-	1.319-	1.195-	1.247-	1.554-	1.351-	1.017∗	1.669-	0.487∗	0.657∗	1.341-	1.052∗
		ResNet32	1.452	0.874-	1.361-	1.079-	1.407-	1.376-	1.534-	1.344-	1.149-	1.675-	1.130-	1.421-	0.346∗	1.030-	0.952-
	Caltech-101	ResNet20	3.851	2.727∗	3.390∗	2.864∗	3.230∗	3.966-	3.202∗	3.770-	3.389∗	3.576-	4.144∗	2.823∗	2.036∗	3.844-	3.270∗
		ResNet32	3.706	2.773∗	2.222∗	2.580∗	2.553∗	3.426∗	3.026∗	3.163∗	3.193∗	3.024∗	3.537-	2.795∗	2.534∗	3.350∗	2.343∗
Select 7.5%	MNIST	LeNet5	4.921	3.036∗	3.120∗	3.149∗	3.231∗	4.863-	3.167∗	5.102∗	3.461∗	3.048∗	4.431∗	3.256∗	4.749∗	3.473∗	3.161∗
		LeNet1	4.940	3.127∗	3.178∗	3.164∗	3.217∗	4.916-	3.833∗	3.977∗	4.031∗	4.001∗	4.265∗	2.453∗	4.155∗	4.098∗	3.257∗
	Fashion	LeNet1	4.102	2.984∗	2.862∗	2.749∗	3.050∗	4.100-	3.620∗	3.565∗	3.774∗	3.315∗	3.322∗	2.583∗	2.899∗	3.594∗	2.972∗
		ResNet20	5.146	2.649∗	2.494∗	2.527∗	2.667∗	5.068-	4.503∗	4.768∗	4.790-	4.755∗	4.769∗	3.963∗	3.310∗	5.001-	4.297∗
	CIFAR	VGG16	3.799	2.685∗	2.703∗	2.687∗	-1.889∗	3.473-	3.482∗	3.554-	3.597∗	3.473∗	3.351∗	2.895∗	2.294∗	3.486∗	2.431∗
		ResNet20	2.426	2.365-	1.989∗	1.692∗	1.925-	2.651-	2.206-	2.441-	2.564-	2.910-	2.208-	2.412-	1.302∗	2.589-	2.098-
	SVHN	LeNet5	3.712	1.794∗	1.885∗	1.752∗	1.809∗	3.299∗	2.822∗	2.940∗	3.148∗	2.854∗	3.064∗	1.505∗	2.100∗	2.842∗	1.835∗
		VGG16	4.483	2.706∗	2.715∗	2.651∗	2.836∗	4.325-	3.758∗	4.165∗	3.931∗	3.660∗	3.520∗	3.237∗	1.749∗	3.859∗	2.795∗
	CIFAR-100	ResNet20	1.967	1.718-	1.468∗	1.828-	1.253∗	1.774-	1.704-	1.715-	1.781-	1.796-	1.855-	1.636-	1.203∗	1.502∗	1.596-
		ResNet32	2.325	1.308∗	1.758-	1.660∗	1.732-	2.286-	1.840-	1.757-	1.638-	1.684-	1.084∗	1.714∗	0.779∗	1.508∗	1.194∗
	Caltech-101	ResNet20	4.808	3.471∗	4.046∗	3.774∗	3.823∗	4.643-	4.128∗	4.584-	4.276∗	4.289-	5.143∗	3.626∗	2.806∗	4.511∗	3.824∗
		ResNet32	4.573	3.329∗	3.320∗	2.987∗	2.936∗	3.828∗	3.842∗	4.096∗	3.939∗	3.757∗	4.334∗	3.571∗	3.193∗	4.251∗	3.272∗
Select 10%	MNIST	LeNet5	5.301	3.396∗	3.487∗	3.483∗	3.547∗	5.266-	3.592∗	5.514∗	3.904∗	3.499∗	4.819∗	3.598∗	5.273-	3.996∗	3.543∗
		LeNet1	5.344	3.547∗	3.584∗	3.504∗	3.526∗	5.445-	4.421∗	4.498∗	4.560∗	4.455∗	4.774∗	2.760∗	4.628∗	4.532∗	3.679∗
	Fashion	LeNet1	4.484	3.306∗	3.391∗	3.283∗	3.612∗	4.491-	4.211∗	3.949∗	4.279∗	3.678∗	3.849∗	3.007∗	3.275∗	4.217∗	3.398∗
		ResNet20	5.825	3.008∗	3.197∗	3.145∗	3.193∗	5.718-	5.384∗	5.282∗	5.458-	4.981∗	5.252∗	4.834∗	3.768∗	5.447-	4.568∗
	CIFAR	VGG16	4.105	3.049∗	3.099∗	3.057∗	3.152∗	4.151-	4.006-	4.185-	3.994-	3.971-	4.033-	3.295∗	2.515∗	4.039-	3.017∗
		ResNet20	3.697	2.271∗	2.359∗	2.680∗	2.362∗	2.785∗	2.803∗	2.704∗	3.101-	3.081-	3.246∗	2.797∗	1.692∗	3.683-	2.365∗
	SVHN	LeNet5	4.476	2.476∗	2.415∗	2.316∗	2.414∗	4.091∗	3.632∗	3.521∗	3.737∗	3.613∗	3.795∗	1.954∗	2.618∗	3.642∗	2.299∗
		VGG16	4.937	3.103∗	3.093∗	3.040∗	3.158∗	4.640∗	4.320∗	4.604∗	4.372∗	4.129∗	4.000∗	3.650∗	2.011∗	4.330∗	3.084∗
	CIFAR-100	ResNet20	2.426	2.012-	1.742∗	1.672∗	1.924∗	2.431-	1.962∗	2.501-	2.317-	2.024-	2.141-	1.980-	1.514∗	2.102-	2.275-
		ResNet32	2.321-	1.748∗	2.091-	2.349-	2.178-	2.013-	2.044-	2.140-	2.041-	2.686-	1.369∗	1.963-	1.291∗	1.686-	2.244-
	Caltech-101	ResNet20	5.514	3.896∗	4.824∗	4.387∗	4.180∗	5.206∗	4.889∗	5.244∗	5.057∗	5.039-	5.608-	4.236∗	3.546∗	5.365-	4.359∗
		ResNet32	5.207	3.654∗	3.685∗	3.599∗	3.715∗	4.578∗	4.540∗	4.822∗	4.729-	4.429∗	5.103-	4.180∗	3.893∗	5.154-	3.872∗

1. $* \; p < 0.05, - \; p > 0.05$.

while ATS, Max$p$, and DeepGini perform the best in 4.17%, 20.83%, and 2.08% of cases, respectively. Interestingly, the active learning approaches show satisfactory failure detection performance on the CIFAR-100 dataset, but they significantly underperform compared to RTS on other datasets. Upon further analysis, we discover that the tested DNN's performance on the CIFAR-100 dataset is worse than on other datasets. For example, the accuracy of ResNet-20 on the CIFAR-100 dataset is only 51.46%, whereas it can reach 75.61% on CIFAR-10. A plausible explanation posits that when model performance is suboptimal, test inputs exhibiting higher information entropy potentially provoke aberrant behavior in the DNN. Overall, our experiments clearly indicate that active learning methods are not as effective in identifying misclassified instances. This can be attributed to the fact that active learning methods are primarily designed for model training rather than testing purposes, focusing on data that provides more information for the model. However, informative data does not necessarily lead to more incorrect classifications by the model. Moreover, the experimental results indicate that DMsp can improve the failure detection results of existing TS methods in noisy scenarios, but the improvement is very limited and does not exceed 2%.

We further conduct statistical analysis for each selection rate to investigate whether RTS significantly outperforms all the baseline methods by conducting the Wilcoxon signed-rank test [38] at the significance level 95%. Meanwhile, the non-parametric Cliff's delta effect size is used to evaluate the amount of the difference[3] between the two approaches. Specifically, for each selection rate, we compute the p-value of RTS with respect to each baseline, based on the FD_{ratio} results for the 12 DNN&Dataset configurations (6 subjects $\times$ 2 models). We found that all the p-values are smaller than 0.05, indicating that RTS significantly outperforms all the compared approaches in terms of failure detection metric in statistics. The effect-size results show that compared with

3. We use the following mapping for the values of the delta that are less than 0.147, between 0.147 and 0.33, between 0.33 and 0.474 and above 0.474 as "*Negligible (N)*", "*Small (S)*", "*Medium (M)*", "*Large (L)*" effect size, respectively [64]



(a) $Selection_{rate} = 0.025$ (b) $Selection_{rate} = 0.05$ (c) $Selection_{rate} = 0.075$ (d) $Selection_{rate} = 0.01$

Fig. 6. The average failure detection rate (%) of RTS and the three variants for each configuration.

neuron-coverage-based, other TS methods (such as CES and LSA), and CLUE, RTS can bring large performance differences regardless of selection rate, dataset, and DNNs under test. Besides, RTS has a medium or small average performance difference in contrast to DeepGini, Max$p$, and ATS.

☛ We conclude that RTS has an outstanding capability of detecting more failures within a limited selection size. For example, when selecting 2.5% test input, RTS achieves an improvement of 9.37%-176.75% over other baseline methods in terms of failure detection.

6.2 Answer to RQ2: Retraining Effectiveness

To address RQ2, we merge the test inputs selected by different TS methods with the original training set to evaluate the effectiveness of DNN optimization. Following the previous work [15], we collect the accuracy improvement results for four selection ratios (2.5%, 5%, 7.5%, 10%). The accuracy improvement results are provided in Table 6, where the best optimization effectiveness (OE) value (as defined in Equation 11) is highlighted in red, and the second best OE value is highlighted in blue for each DNN&Dataset configuration. Each TS method has 15 OE results (5 candidate sets $\times$ 3 experiment repetitions) under each DNN&Dataset configuration. The table presents the average performance of these 15 OE results. Additionally, p-values are computed to determine if there are statistically significant differences among the investigated methods based on the 15 OE results. A "$*$" is used to mark the corresponding accuracy improvement value if the p-value is less than 0.05; otherwise, a "-" is used.

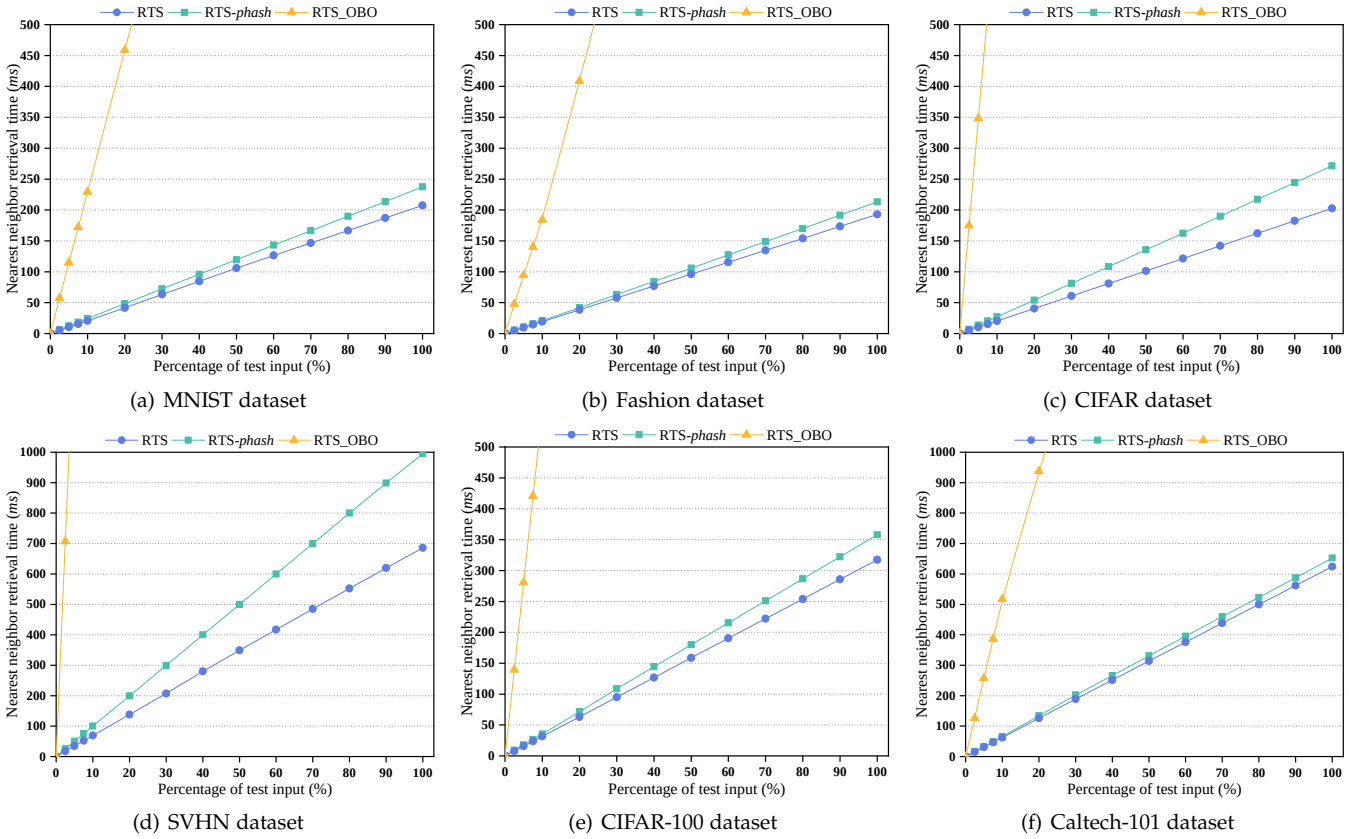
From the analysis of Table 6, it is evident that most TS methods exhibit superior performance compared to random sampling in terms of average accuracy improvement. Notably, for neuron-coverage-based selection techniques, there are cases where the retraining results are even worse than random sampling with the same test input size. Regarding the comparison among different TS methods, RTS outperforms the other methods in 56.25% of cases (27 out of 48), making it the most effective approach. ATS achieves the best result in 8.33% of cases (4 out of 48), while DeepGini performs the best in only 2.08% of cases (1 out of 48). Additionally, VS demonstrates better DNN accuracy improvement compared to DeepGini due to its more accurate uncertainty calculation. VS achieves the best result in 12.5% of cases (6 out of 48), but it still falls short compared to RTS, which achieves the best or second-best result in 79.17% of cases. Statistical analysis reveals that, compared with RTS, most baseline methods perform worse, especially the neuron-coverage-based and other types of TS methods. Apart from ATS and VS, RTS significantly outperforms the other 12 baseline methods in more than 28 cases. When comparing with VS and ATS, RTS demonstrates significantly better DNN accuracy improvement results in 27 and 15 instances, respectively. Notably, upon contrasting Max$p$ and CLUE, it becomes apparent that Max$p$ markedly surpasses CLUE in failure detection. This assertion is substantiated by statistical analysis, revealing a considerable disparity irrespective of selection rate and DNN&Dataset configurations. Nevertheless, in model optimization, CLUE accrues best or second-best outcomes in nine instances, outstripping Max$p$, which secures three second-best results. This promising performance by CLUE in model optimization necessitates a deeper examination into additional factors that potentially influence model optimization. Consequently, we ardently advocate for further empirical studies, urging researchers to diligently explore the interplay between distinct test input characteristics and their consequent impact on model optimization.

☛ We conclude that RTS has an outstanding capability of improving the performance of the pre-trained model within a limited selection size.

6.3 Answer to RQ3: Ablation Experiment

6.3.1 *Answer to RQ3.1*

Figure 6 presents the FD_{ratio} results of RTS and its three variants across different DNN&Dataset configurations and $Selection_{rate}$ values. The $Selection_{rate}$ refers to the ratio of selected test inputs and FD_{ratio} represents the failure detection rate, calculated by Equation 10. Three variants, RTS-1, RTS-2, and RTS-3, are constructed by excluding Component 1, Component2, and Component 3, respectively. This allows us to evaluate the individual contributions of each component to the effectiveness of RTS. The elimination of any single component within RTS typically precipitates a deterioration in failure detection performance, albeit to differing extents across various scenarios. Among the three components, the probability-tier-matrix-based selection metric contributes the most to RTS's performance improvement, as the performance of RTS-3 has the worst performance compared to RTS. RTS performs better than RTS-1 in most cases, which indicates the noise-resilient component (i.e., Component 1) effectively



(a) MNIST dataset (b) Fashion dataset (c) CIFAR dataset

(d) SVHN dataset (e) CIFAR-100 dataset (f) Caltech-101 dataset

Fig. 7. The retrieval time of RTS and the two variants for each dataset.

improves the robustness of RTS in candidate sets with noise. The paired Mann Whitney-Wilcoxon Test [65] is used to verify the significance of the performance differences. All the p-values between RTS and RTS-1, as well as RTS and RTS-3, are less than 0.05, indicating that RTS significantly outperforms these two variants. Although the performance difference between RTS and RTS-2 is not significant, RTS still obtains similar or better results in most cases. For instance, when $selection_{rate} = 0.025$, RTS achieves better FD_{ratio} values for 7 out of 12 DNN&Dataset combinations, and similarly for the other selection rates. In instances where RTS-2 outperforms RTS, we hypothesize that the discrepancy arises from Component 2 misclassifying failure-revealing test inputs as successful ones. This issue could potentially be mitigated by amplifying the k value. Above-mentioned results indicate that the noise filtering component, the voting mechanism, and the novel test metric are useful and effective for improving RTS's failure detection performance.

☛ Overall, when each of the three components of the RTS is removed, the failure detection performance of the RTS is typically reduced to varying degrees, with Component 1 and Component 3 significantly affecting the performance of the RTS.

6.3.2 Answer to RQ3.2

In Section 5.4.3, two variants of the RTS algorithm, namely RTS-OBO and RTS-*p*hash, are introduced to evaluate the impact of two pivotal steps in the nearest neighbor retrieval process. To verify that RTS effectively preserves the global similarity matching capability, 20% noise data comprising irrelevant data, synthetic meaningless data, and crashed data, is integrated into the original test set. This enables us to delve into the performance of these variants in noise data identification. Specifically, the efficiency of RTS and its variants in retrieving the nearest neighbors for different percentages (n) of test inputs from each dataset's candidate set is assessed. Chosen values for (n) encompass 2.5%, 5.0%, 7.5%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, and 100%. Ultimately, we report the accuracy results with which each variant is capable of detecting noise within the candidate set.

Figure 7 presents the nearest neighbor retrieval time for RTS and its variants. Each subfigure corresponds to a specific dataset, where the x-axis represents the percentage of test inputs from the candidate test set, and the y-axis indicates the time taken to retrieve the nearest neighbors for those test inputs. Based on the data in Figure 7, we make the following observations: 1) ***RTS vs. RTS-OBO:*** RTS-OBO always requires more retrieval time than RTS for the same number of test inputs. This is because RTS-OBO needs to compare the similarity of each test input with all training data one by one. This disadvantage becomes prominent when both the training data and the candidate test sets are relatively large. 2) ***RTS vs. RTS-phash:*** RTS consistently outperforms RTS-*p*hash in terms of retrieval time across all datasets and test input percentages. When the number of test inputs is small, such as only 2.5%, the retrieval time of RTS is comparable to that of RTS-*p*hash. We analyze that this is due to the overhead incurred by computing the hash values of both the test inputs and training data. However, as the number of test inputs increases, the difference in retrieval time becomes significantly larger. This highlights the

effectiveness of the p-hash algorithm in accelerating nearest neighbor retrieval, particularly when dealing with a large number of test inputs.

TABLE 7
The accuracy results of RTS *vs.* its variants in identifying noise

Method	MNIST	Fashion	CIFAR	SVHN	CIFAR-100	Caltech-101
RTS-OBO	0.9620	0.8517	0.8828	0.9265	0.8681	0.8628
RTS-$p$hash	0.9498	0.8550	0.8942	0.9442	0.8998	0.8597
RTS	0.9490	0.8562	0.8898	0.9449	0.8987	0.8592

Table 7 further elucidates the accuracy of RTS and its two variants in identifying noise. The experimental results reveal that approximate nearest neighbor retrieval, as exhibited by RTS-$p$hash and RTS, consistently achieves noise identification accuracy comparable to that of exact nearest neighbor retrieval (i.e., RTS-OBO). Notably, RTS-OBO does not always secure the best performance. For instance, both RTS-$p$hash and RTS yield superior accuracy results on the SVHN and CIFAR-100 datasets. Analysis indicates that this occurrence might stem from the degree of "fault tolerance" to proximity retrievals, given the challenges in assuring that SSIM furnishes a wholly accurate similarity assessment. Additionally, RTS demonstrates a noise detection performance strikingly similar to that of RTS-$p$hash, with a mere average accuracy difference of 0.0008. This substantiates our descriptions in Section 4.3.1 that, although p-hash predominantly concentrates on local similarity, it effectively maximizes the retention of image pairs with substantial global similarity. Considering the notable temporal advantage and minimal performance degradation afforded by p-hash, it is concluded that RTS presents the most cost-effective performance.

☛ In summary, the experimental results underscore that RTS facilitates an expedient retrieval of nearest neighbors, eclipsing its variants RTS-OBO and RTS-$p$hash in retrieval time efficiency. Moreover, this reduction in computational demand exerts only a marginal impact on the ability to pinpoint matches in global information, positioning RTS as the best selection that furnishes the optimal cost-effectiveness.

7 Discussion

7.1 The Performance of RTS in Detecting Noise Input

Baselines. To mitigate the effects of noise, RTS employs a novel SSIM-based noise filtering component that identifies and filters out invalid test inputs. In order to evaluate the performance of RTS in detecting noise, we compare it with three related techniques: *Embedding$_{sim}$*, *One-class SVM* [66], and *HDBSCAN* algorithm [67]. Here is a brief description of each technique: 1) ***Embedding$_{sim}$*** shares a similar approach with RTS but uses the feature embedding provided by the tested model as the similarity metric. It calculates the similarity between test inputs based on their embeddings to identify potential noise. 2) ***One-class SVM*** identifies anomalies or noise in a dataset by learning the boundaries of the majority class. It considers data points that fall outside of these boundaries as potential noise. 3) ***HDBSCAN*** is a density-based clustering algorithm that has been utilized in prior research related to TS [68]. It assigns a probability of being a cluster member to each data point, classifying any point with a low probability as noise.

Data Collection. We utilize the datasets introduced in Section 5.1 and apply the same construction method described in Section 5.2 to generate additional valid test inputs, besides the original test set. To assess the robustness of RTS and the three noise detection methods, we introduce 20% of invalid data using three data pollution techniques [15]: irrelevant data (IR), meaningless synthetic data (RG), and crashed data (CR). Subsequently, we evaluate the ability of such methods to precisely identify noise data (*Positive: noise data*, *Negative: valid test input*).

Evaluation Metrics. We use accuracy, precision, recall, F1-score, and area under the precision recall curve (AUC) [69] as evaluation measures.

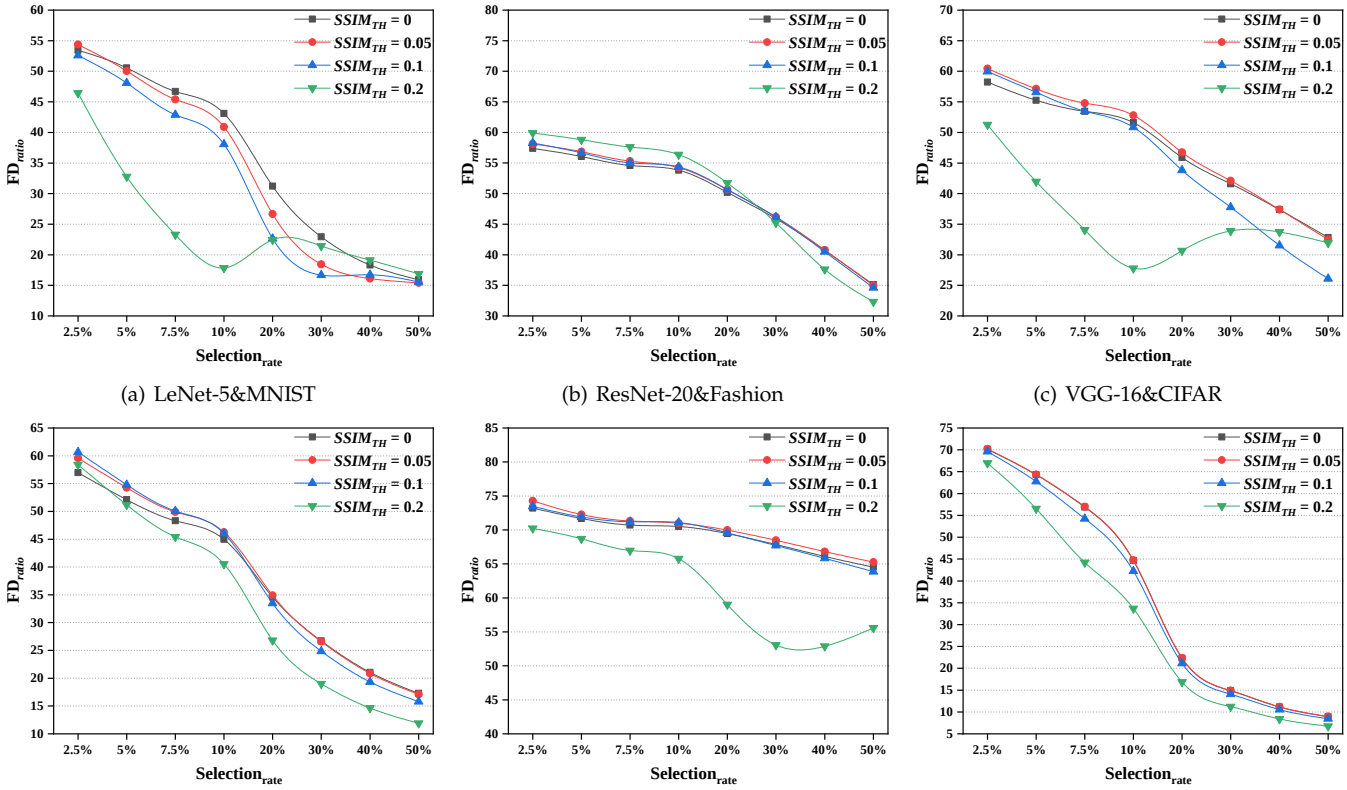
Table 8 presents a comprehensive overview of the performance metrics for each method, including accuracy, precision, recall, F1-score, and AUC. The experimental results reveal valuable insights: 1) **Embedding$_{\text{sim}}$**: This method usually has poor noise detection performance, although it occasionally achieves a precision exceeding 0.90. In complex datasets like VGG-16&CIFAR, the precision, recall, and F1-score of Embedding$_{\text{sim}}$ are all 0.0 in the IR and CR categories. Upon analysis, we attribute this to two factors: (1) The tested model is optimized based on numerous valid test inputs, allowing it to effectively recognize and differentiate between the feature embeddings of valid test inputs with different categories, but not those of noise. (2) Noisy data often possesses irregular and diverse characteristics that the neural network may struggle to capture effectively. 2) **HDBSCAN**: HDBSCAN achieves the best recall values in 13 out of 36 cases, ranking second overall. This method successfully identifies and eliminates a significant amount of noise by identifying data with a low probability of being a cluster member. However, this approach also results in a large number of valid test inputs being filtered out. For instance, in the LeNet-5&SVHN dataset, HDBSCAN filter out all irrelevant test inputs (IR noise types), but its precision is only 0.1683. Moreover, manual examination of the noisy data detected by HDBSCAN reveals that many of them are actually failure-revealing test inputs. This suggests that HDBSCAN discards many valuable failure-revealing test inputs while filtering noise, which could potentially reduce the failure detection performance of the TS method during subsequent test selection. 3) **One-class SVM**: Similar to Embedding$_{\text{sim}}$ and HDBSCAN, the performance of One-class SVM varies across different metrics and datasets. For example, when identifying noisy data in LeNet-1&Fashion, One-class SVM achieves better AUC results than Embedding$_{\text{sim}}$ and HDBSCAN, regardless of the pollution types. However, for the ResNet-20&Fashion, One-class SVM exhibits inferior AUC performance compared to the other two baselines in CR and IR categories, and it also performs worse than Embedding$_{\text{sim}}$ on RG. This can be explained that when training One-class SVM, information is extracted from the last hidden layer of each model as representations of the training samples. Even for the same dataset, the different architectures of tested models result in different extracted features, leading to differences in the effectiveness of noise filtering to some extent. Therefore, for the aforementioned outlier/noise detection methods, including Embedding$_{\text{sim}}$, One-class SVM, and HDBSCAN,

TABLE 8
RTS *vs.* outlier detection methods in identifying noise

Dataset	Noise	Method	Acc.	Prec.	Rec.	F1.	AUC	Dataset	Noise	Method	Acc.	Prec.	Rec.	F1.	AUC
LeNet5_MNIST	CR	$Embedding_{sim}$	0.8309	0.3077	0.0115	0.0222	0.5032	LeNet1_MNIST	CR	$Embedding_{sim}$	0.8179	0.2263	0.0383	0.0654	0.5061
		One-class SVM	0.7893	0.0562	0.0168	0.0258	0.4803			*One-class SVM*	0.7569	0.0784	0.0426	0.0552	0.4712
		HDBSCAN	0.1201	0.0737	0.3700	0.1229	0.2201			*HDBSCAN*	0.1571	0.1559	**0.9193**	0.2666	0.4620
		RTS	**0.8379**	**0.5086**	**0.8164**	**0.6267**	**0.8293**			*RTS*	**0.8611**	**0.5628**	0.7476	**0.6422**	**0.8157**
	IR	$Embedding_{sim}$	0.8299	0.0060	0.0001	0.0002	0.4980		IR	$Embedding_{sim}$	0.8251	0.3723	0.0721	0.1208	0.5239
		One-class SVM	0.7867	0.0049	0.0014	0.0021	0.4726			*One-class SVM*	0.7658	0.1611	0.0963	0.1205	0.4980
		HDBSCAN	0.1399	0.0971	0.5013	0.1627	0.2844			*HDBSCAN*	0.1450	0.0803	0.3949	0.1334	0.2450
		RTS	**0.8517**	**0.5327**	**0.8990**	**0.6690**	**0.8706**			*RTS*	**0.8792**	**0.5957**	**0.8560**	**0.7025**	**0.8699**
	RG	$Embedding_{sim}$	0.8236	0.0224	0.0014	0.0026	0.4947		RG	$Embedding_{sim}$	0.8186	0.1326	0.0160	0.0286	0.4975
		One-class SVM	0.7865	0.0018	0.0005	0.0008	0.4721			*One-class SVM*	0.7657	0.1600	0.0955	0.1196	0.4976
		HDBSCAN	0.1280	0.1167	0.6443	0.1976	0.3345			*HDBSCAN*	0.0346	0.0101	0.0493	0.0167	0.0405
		RTS	**0.8683**	**0.5587**	**0.9986**	**0.7165**	**0.9204**			*RTS*	**0.9029**	**0.6321**	**0.9981**	**0.7740**	**0.9410**
LeNet1_Fashion	CR	$Embedding_{sim}$	0.8397	**0.9501**	0.0405	0.0777	0.5200	ResNet20_Fashion	CR	$Embedding_{sim}$	**0.8767**	0.7067	0.4448	**0.5459**	**0.7039**
		One-class SVM	0.8348	0.5070	0.3111	**0.3856**	**0.6253**			*One-class SVM*	0.7939	0.0256	0.0064	0.0102	0.4789
		HDBSCAN	0.5203	0.2083	**0.6705**	0.3178	0.5804			*HDBSCAN*	0.5469	0.2452	**0.8273**	0.3783	0.6590
		RTS	**0.8570**	0.7770	0.1995	0.3175	0.5940			*RTS*	0.8534	**0.7961**	0.1620	0.2692	0.5769
	IR	$Embedding_{sim}$	0.8341	**1.0000**	0.0048	0.0095	0.5024		IR	$Embedding_{sim}$	0.8240	0.1394	0.0109	0.0202	0.4987
		One-class SVM	**0.9479**	0.7659	**0.9899**	**0.8636**	**0.9647**			*One-class SVM*	0.7930	0.0041	0.0010	0.0016	0.4762
		HDBSCAN	0.4638	0.2035	0.7605	0.3210	0.5825			*HDBSCAN*	0.3985	0.1717	**0.6825**	**0.2744**	0.5121
		RTS	0.8373	0.5866	0.0813	0.1427	0.5349			*RTS*	**0.8356**	**0.5705**	0.0551	0.1005	**0.5234**
	RG	$Embedding_{sim}$	0.8332	0.3333	0.0008	0.0015	0.5002		RG	$Embedding_{sim}$	**0.9296**	0.8538	**0.6966**	**0.7673**	**0.8364**
		One-class SVM	0.8114	0.3608	0.1708	0.2318	0.5551			*One-class SVM*	0.8823	0.6884	0.5363	0.6029	0.7438
		HDBSCAN	0.1055	0.0179	0.0813	0.0294	0.0958			*HDBSCAN*	0.4930	0.1252	0.3410	0.1831	0.4322
		RTS	**0.8911**	**0.8758**	**0.4038**	**0.5527**	**0.6962**			*RTS*	0.8907	**0.9029**	0.3859	0.5407	0.6888
Vgg16_CIFAR	CR	$Embedding_{sim}$	0.8427	**0.9536**	0.0591	0.1113	0.5293	ResNet20_CIFAR	CR	$Embedding_{sim}$	0.8333	0.0000	0.0000	0.0000	0.5000
		One-class SVM	0.7934	0.0092	0.0023	0.0036	0.4770			*One-class SVM*	0.7865	0.1663	0.0700	0.0985	0.4999
		HDBSCAN	0.4302	0.1157	0.3643	0.1757	0.4038			*HDBSCAN*	0.2838	0.0456	0.1656	0.0716	0.2366
		RTS	**0.9574**	0.8098	**0.9725**	**0.8837**	**0.9634**			*RTS*	**0.9526**	**0.7885**	**0.9776**	**0.8729**	**0.9626**
	IR	$Embedding_{sim}$	**0.8326**	0.0000	0.0000	0.0000	**0.4996**		IR	$Embedding_{sim}$	**0.8333**	0.0000	0.0000	0.0000	0.5000
		One-class SVM	0.7931	0.0005	0.0001	0.0002	0.4759			*One-class SVM*	0.8151	**0.4078**	**0.2418**	**0.3036**	**0.5858**
		HDBSCAN	0.1881	**0.0986**	**0.4754**	**0.1633**	0.3030			*HDBSCAN*	0.2884	0.0450	0.1615	0.0703	0.2376
		RTS	0.7963	0.0272	0.0064	0.0103	0.4804			*RTS*	0.7910	0.0296	0.0080	0.0126	0.4778
	RG	$Embedding_{sim}$	0.8321	0.0000	0.0000	0.0000	0.4992		RG	$Embedding_{sim}$	0.8411	**0.9820**	0.0476	0.0908	0.5237
		One-class SVM	0.7960	0.0684	0.0178	0.0282	0.4847			*One-class SVM*	0.8761	0.6338	0.6074	0.6203	0.7686
		HDBSCAN	0.1094	0.0038	0.0168	0.0062	0.0723			*HDBSCAN*	0.1637	0.1002	0.5035	0.1671	0.2996
		RTS	**0.9049**	**0.7422**	**0.6576**	**0.6974**	**0.8060**			*RTS*	**0.8993**	0.7150	**0.6579**	**0.6852**	**0.8027**
ResNet20_Caltech-101	CR	$Embedding_{sim}$	**0.8333**	0.0000	0.0000	0.0000	0.4999	ResNet32_Caltech-101	CR	$Embedding_{sim}$	0.6080	0.2674	**0.7779**	**0.3980**	0.6760
		One-class SVM	0.7918	0.2846	0.1650	0.2089	0.5410			*One-class SVM*	0.6857	0.0026	0.0023	0.0024	0.4123
		HDBSCAN	0.8330	**0.4982**	**0.3254**	**0.3937**	**0.6299**			*HDBSCAN*	0.6510	0.2257	0.4503	0.3006	0.5707
		RTS	0.8290	0.3803	0.0416	0.0751	0.5140			*RTS*	**0.8310**	**0.4240**	0.0409	0.0745	0.5149
	IR	$Embedding_{sim}$	**0.8334**	0.0000	0.0000	0.0000	**0.5000**		IR	$Embedding_{sim}$	0.6091	**0.2980**	**0.9931**	**0.4584**	**0.7627**
		One-class SVM	0.7643	0.0000	0.0000	0.0000	0.4585			*One-class SVM*	0.6854	0.0009	0.0008	0.0008	0.4115
		HDBSCAN	0.3089	**0.1172**	**0.4819**	**0.1885**	0.3781			*HDBSCAN*	0.5029	0.1124	0.2876	0.1616	0.4168
		RTS	0.8221	0.0000	0.0000	0.0000	0.4932			*RTS*	**0.8241**	0.0000	0.0000	0.0000	0.4945
	RG	$Embedding_{sim}$	0.8333	0.0000	0.0000	0.0000	0.4999		RG	$Embedding_{sim}$	0.6227	0.2816	**0.8150**	0.4185	0.6996
		One-class SVM	0.8014	0.3495	0.2228	0.2721	0.5699			*One-class SVM*	0.6870	0.0111	0.0100	0.0106	0.4162
		HDBSCAN	0.8297	0.1778	0.0062	0.0119	0.5002			*HDBSCAN*	0.6172	0.1110	0.1850	0.1387	0.4443
		RTS	**0.9315**	**0.9064**	**0.6569**	**0.7617**	**0.8217**			*RTS*	**0.9336**	**0.9221**	0.6569	**0.7672**	**0.8229**
ResNet20_CIFAR-100	CR	$Embedding_{sim}$	0.8333	0.0000	0.0000	0.0000	0.5000	ResNet32_CIFAR-100	CR	$Embedding_{sim}$	0.8328	0.0345	0.0001	0.0002	0.4997
		One-class SVM	0.7807	0.0205	0.0068	0.0102	0.4711			*One-class SVM*	0.7883	0.0041	0.0011	0.0018	0.4735
		HDBSCAN	0.2677	0.0591	0.2275	0.0938	0.2516			*HDBSCAN*	0.4483	0.1857	**0.6826**	0.2920	0.5421
		RTS	**0.9390**	**0.8998**	**0.7131**	**0.7957**	**0.8486**			*RTS*	**0.9215**	**0.9266**	0.5745	**0.7093**	**0.7827**
	IR	$Embedding_{sim}$	**0.8334**	**1.0000**	0.0001	0.0002	**0.5001**		IR	$Embedding_{sim}$	**0.8335**	**0.6452**	0.0025	0.0050	0.5011
		One-class SVM	0.7849	0.0893	0.0316	0.0467	0.4836			*One-class SVM*	0.7890	0.0190	0.0053	0.0082	0.4755
		HDBSCAN	0.2938	0.0827	**0.3206**	**0.1314**	0.3045			*HDBSCAN*	0.5048	0.2280	**0.8264**	**0.3574**	**0.6334**
		RTS	0.8202	0.0078	0.0006	0.0012	0.4924			*RTS*	0.8258	0.0027	0.0001	0.0002	0.4955
	RG	$Embedding_{sim}$	0.8441	1.0000	0.0649	0.1218	0.5324		RG	$Embedding_{sim}$	0.8330	0.2000	0.0008	0.0015	0.5001
		One-class SVM	0.8583	0.5941	0.4721	0.5262	0.7038			*One-class SVM*	0.8071	0.2951	0.1135	0.1639	0.5296
		HDBSCAN	0.0987	0.0711	0.3651	0.1190	0.2053			*HDBSCAN*	0.4469	0.1439	0.4684	0.2201	0.4555
		RTS	**0.9296**	**0.8922**	**0.6573**	**0.7569**	**0.8207**			*RTS*	**0.9351**	**0.9352**	**0.6564**	**0.7714**	**0.8236**
Vgg16_SVHN	CR	$Embedding_{sim}$	0.8329	0.0959	0.0003	0.0007	0.4999	LeNet5_SVHN	CR	$Embedding_{sim}$	0.8329	0.0959	0.0003	0.0007	0.4999
		One-class SVM	0.8028	0.0000	0.0000	0.0000	0.4817			*One-class SVM*	0.8254	0.4787	0.5351	0.5053	0.7093
		HDBSCAN	0.5612	0.1793	0.4564	0.2575	0.5193			*HDBSCAN*	0.1598	0.1566	0.9213	0.2677	0.4644
		RTS	**0.9823**	**0.9328**	**0.9632**	**0.9478**	**0.9747**			*RTS*	**0.9756**	**0.8725**	**0.9994**	**0.9316**	**0.9851**
	IR	$Embedding_{sim}$	0.8333	0.3333	0.0000	0.0001	0.5000		IR	$Embedding_{sim}$	0.8333	0.3333	0.0000	0.0001	0.5000
		One-class SVM	0.8058	0.0892	0.0180	0.0299	0.4906			*One-class SVM*	0.8343	0.5025	0.5884	0.5420	0.7359
		HDBSCAN	0.6324	0.3106	**0.9883**	**0.4726**	**0.7747**			*HDBSCAN*	0.1766	0.1683	**1.0000**	0.2882	0.5060
		RTS	**0.8718**	**0.8123**	0.3002	0.4384	0.6432			*RTS*	**0.9319**	**0.8346**	0.7372	**0.7829**	**0.8540**
	RG	$Embedding_{sim}$	0.8332	0.0000	0.0000	0.0000	0.4999		RG	$Embedding_{sim}$	0.8332	0.0000	0.0000	0.0000	0.4999
		One-class SVM	0.8377	0.5332	0.2096	0.3009	0.5864			*One-class SVM*	0.7468	0.0981	0.0633	0.0770	0.4734
		HDBSCAN	0.5600	0.1077	0.2251	0.1457	0.4260			*HDBSCAN*	0.1365	0.1367	**0.7870**	0.2330	0.3967
		RTS	**0.9323**	**0.9053**	**0.6631**	**0.7655**	**0.8246**			*RTS*	**0.9198**	**0.8199**	0.6650	**0.7343**	**0.8179**
AVRAGE		$Embedding_{sim}$	0.8338	0.3335	0.0102	0.0193	0.5044	**AVRAGE**		$Embedding_{sim}$	0.8021	0.3094	0.2176	0.1623	0.5683
		One-class SVM	0.8093	0.2075	0.1465	0.1631	0.5441			*One-class SVM*	0.7825	0.2076	0.1676	0.1802	0.5365
		HDBSCAN	0.3662	0.1410	0.4045	0.1839	0.3815			*HDBSCAN*	0.3419	0.1360	0.5363	0.2123	0.4197
		RTS	**0.8811**	**0.6198**	**0.5017**	**0.5088**	**0.7293**			*RTS*	**0.8869**	**0.6295**	**0.5099**	**0.5206**	**0.7361**

the feature embeddings extracted by the tested model greatly influence the effectiveness of these methods.

RTS demonstrates the best performance across all metrics. Among the 36 cases (6 subjects $\times$ 2 models $\times$ 3 data pollution types), RTS outperforms the other methods in 75.00% (27/36) cases for accuracy, 69.44% (25/36) for precision, 47.22% (17/36) for recall, 61.11% (22/36) for F1-score, and 63.89% (23/36) for AUC. The average values of accuracy, precision, recall, F1-score, and AUC for RTS are 0.8840, 0.6246, 0.5058, 0.5147, and 0.7327, respectively, with improvements compared to the baseline methods ranging from 8.07% to 149.68% for accuracy, 94.33% to 350.96% for precision, 7.53% to 344.19% for recall, 159.83% to 466.78% for F1-score, and 35.61% to 82.91% for AUC. RTS filters noise by utilizing the original features of the image instead of relying on information from the hidden layer, which enables RTS to maintain consistent performance even when applied to the same dataset with different models under test. Analyzing Table 8, we observe that the SSIM-based filtering method can distinguish well between noisy and valid test inputs for the MNIST dataset, with an average AUC value of 0.8745. However, for complex datasets such as CIFAR and CIFAR-100, the effectiveness of the SSIM-based filtering method decreases slightly, but it still achieves satisfactory



(a) LeNet-5&MNIST (b) ResNet-20&Fashion (c) VGG-16&CIFAR

(d) VGG-16&SVHN (e) ResNet-32&CIFAR-100 (f) ResNet-20&Caltech-101

Fig. 8. The exploration of $SSIM_{th}$ setting.

performance with an average AUC value of 0.7482 and 0.7106, respectively. It is worth mentioning that an AUC of 0.7 is considered as a promising performance score, as stated in previous studies [70], [71].

TABLE 9
The average failure detection rate (%) for each selection rate in noiseless candidate set.

Baselines	Select 2.5%	Select 5.0%	Select 7.5%	Select 10.0%
ATS	69.56	66.54	63.97	62.07
Gini	75.91	73.04	70.71	68.84
CES	27.98	28.63	28.73	28.64
LSA	52.96	51.38	50.34	48.72
Maxp	75.60	72.50	70.19	68.05
EDrop	75.01	71.84	69.84	67.80
CLUE	40.27	40.14	39.47	38.70
VS	56.45	56.28	55.74	55.25
DMsp	75.53	72.46	70.08	68.28
NAC	26.76	26.64	26.49	26.64
NBC	27.52	27.20	27.22	27.08
SNAC	26.50	26.97	27.05	26.87
TKNC	28.90	28.15	27.96	27.72
Random	26.78	26.48	26.64	26.87
RTS	71.84	69.16	66.60	64.60

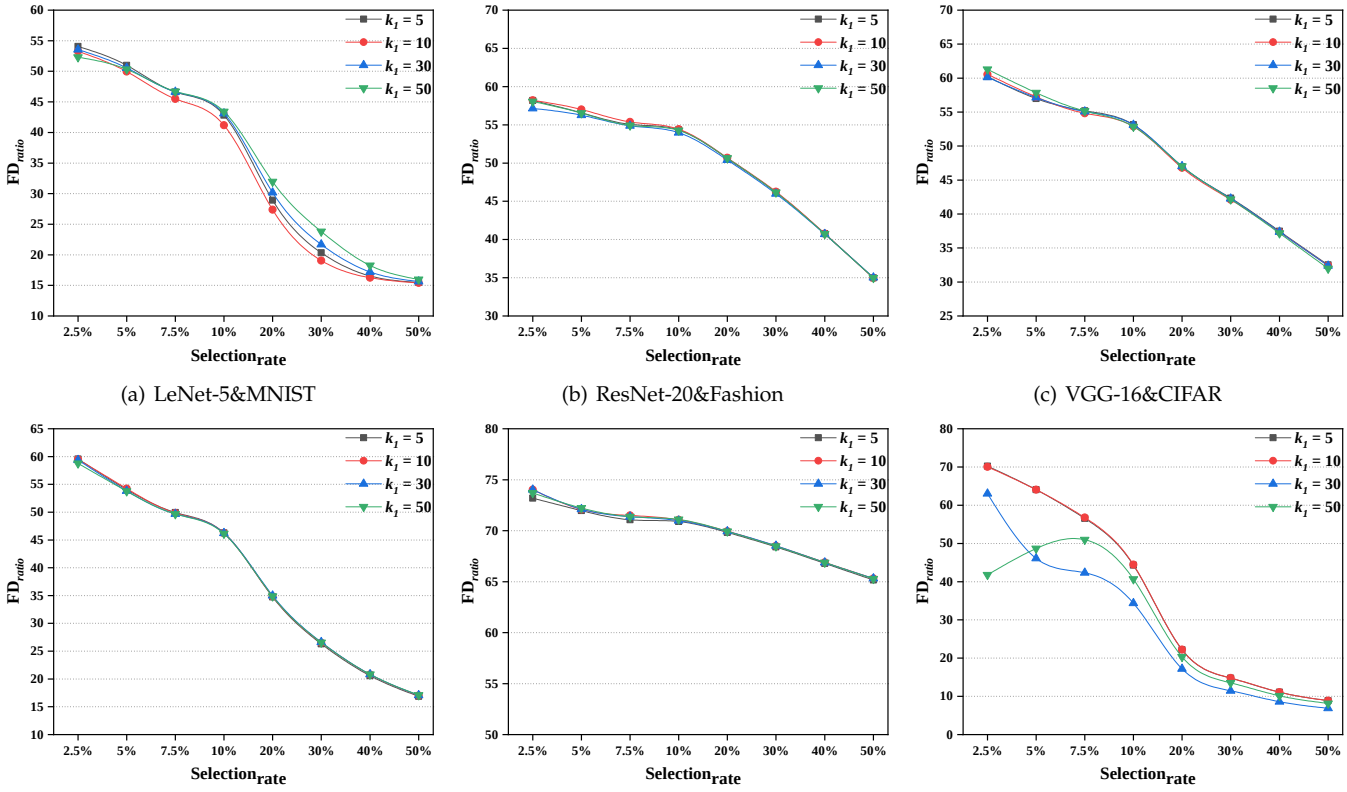
7.2 RTS Performance in Noiseless Candidate set

In this section, we focus on evaluating the performance of RTS and its baseline methods specifically in the noiseless candidate set. Our evaluation considers both the failure detection capability and the model optimization capability. Due to space limitations, we provide average results for each TS method across 12 DNN&Dataset configurations at each selection rate. For more detailed results, please refer to our open-source replication package[4]. Table 9 provides the average failure detection rate for each selection rate. From the table, we can observe that among the 15 TS methods, RTS ranks fifth in terms of failure detection, regardless of the selection rate. However, DeepGini stands out by being able to identify the highest number of failure-revealing test inputs. It is understandable, because DeepGini and Max*p* methods prioritize test inputs with the highest uncertainty. In contrast, RTS aims to diversify the selected test input, which may result in a slight loss in the failure detection performance. This also applies to state-of-the-art ATS, which is guided by diversity. Additionally, RTS incorporates a component for

TABLE 10
The DNNs' accuracy improvement (%) performance for each selection rate in noiseless candidate set.

Baselines	Select 2.5%	Select 5.0%	Select 7.5%	Select 10.0%
ATS	2.51	3.42	4.17	4.67
Gini	2.36	3.55	4.07	4.55
CES	1.64	2.19	2.80	3.19
LSA	2.00	2.78	3.27	3.94
Maxp	2.46	3.54	4.16	4.89
EDrop	2.30	3.40	4.04	4.47
CLUE	2.22	3.12	3.53	4.10
VS	2.18	3.31	3.84	4.39
DMsp	2.42	3.32	3.96	4.64
NAC	1.42	2.26	2.71	3.08
NBC	1.53	2.32	2.78	3.23
SNAC	1.35	2.15	2.75	3.16
TKNC	1.53	2.32	2.74	3.30
Random	1.57	2.26	2.88	3.42
RTS	2.51%	3.50%	4.06%	4.70%

4. https://github.com/swf1996120/RTS



(a) LeNet-5&MNIST (b) ResNet-20&Fashion (c) VGG-16&CIFAR

(d) VGG-16&SVHN (e) ResNet-32&CIFAR-100 (f) ResNet-20&Caltech-101

Fig. 9. The exploration of k setting.

filtering noise, which may misclassify valid test inputs as noise, especially for noiseless candidate sets. This dynamic is similarly mirrored in DMsp, where the incorporation of a pre-processing phase aimed at filtering out-of-distribution (OOD) data inadvertently precipitates a sub-optimal performance relative to DeepGini. However, selecting more failure-revealing test inputs does not necessarily indicate better model optimization capabilities. When examining Table 9 and Table 10, we find that DeepGini do not show significant accuracy gains after retraining, even though it can select a large number of failure-revealing test inputs. As explained in Section 3.1, using biased failure-revealing test inputs does not enhance model performance but rather results in biased DNNs. This issue can be addressed by selecting diverse test inputs, which is a focus of RTS.

7.3 Exploration of RTS Parameter Settings

As explained, the performance of the RTS is influenced by two import parameters: $SSIM_{th}$ and k. The value of $SSIM_{th}$ determines the threshold for filtering noise data in Component 1 based on the SSIM. On the other hand, k determines the number of nearest neighbors considered in both Component 1 and Component 2. Both of these parameters are crucial in the noise filtering process, and the value of k also impacts the number of successful test inputs identified by Component 2. This, in turn, affects the final failure detection rate of the RTS. Therefore, this section focuses on exploring the influence of different $SSIM_{th}$ and k parameter values on the RTS. To conduct the analysis, we follow the same data construction approach as in RQ1 and RQ2. Due to space limitation, we select 6 DNN&Dataset configurations, i.e., LeNet-5&MNIST, ResNet-20&Fashion, VGG-16&CIFAR, VGG-16&SVHN, ResNet-32&CIFAR-100, and ResNet-20&Caltech-101. These configurations represent cases where the selected DNNs achieve the highest accuracy values in their respective datasets. All other results exhibit a similar trend and can be accessed online[5] for further examination. To cover a wide range of scenarios, we set $SSIM_{th}$ to four different values: 0, 0.05, 0.1, and 0.2. Similarly, we vary the value of k across four levels: 5, 10, 30, and 50. By examining the performance results obtained with different parameter values, we gain insights into the impact of these parameters on the RTS's performance.

Figure 8 and Figure 9 present the failure detection rate results of RTS for the different parameter values. The data presented in the figures allows us to make several observations: 1) As $SSIM_{th}$ increases, the difference in FD_{ratio} of RTS becomes significant. Generally, lower values of $SSIM_{th}$ correspond to better fault detection performance, particularly for the LeNet-5&MNIST. When a higher SSIM threshold is employed, RTS has greater confidence in identifying valid test inputs, but it may also discard some failure-revealing test inputs that could trigger unexpected behavior of the tested model, despite exhibiting visual differences from the training samples. Comparing $SSIM_{th} = 0$ and $SSIM_{th} = 0.05$, their performance is comparable. For example, $SSIM_{th} = 0.05$ achieves optimal results with VGG-16&CIFAR but it performs worse than $SSIM_{th} = 0$ when applied to LeNet-5&MNIST. 2) Increasing the value of k can improve the precision of identifying noise and successful test inputs. Overall, there is a slight performance

5. https://github.com/swf1996120/RTS

improvement with increasing k, although the difference is not significant. However, for the Caltech-101 dataset, larger k values actually reduce the failure detection performance of RTS. For example, when $k = 50$ and $Selection_{rate} = 2.5\%$, the FD_{ratio} results of RTS are significantly worse than those of RTS with $k = 5, 10$. This is because some classes in the training set have fewer than 30 data points. When k exceeds the maximum number of similar instances for a test input (i.e., the number of training data points in the same class), the average SSIM score obtained in Component 1 is low, leading to the test input being filtered out as noise.

Based on the above, considering the potential labeling errors in the training dataset, this study opts to set $SSIM_{th} = 0.05$, bolstering the RTS's robustness in noise filtering. This choice effectively mitigates the impact of inaccuracies within the training corpus and ensures the reliability of the noise filtering process in the RTS. Given that a higher k value introduces additional time overhead for RTS, we have selected $k = 5$ in this paper. When testing resources are sufficient, testers can consider employing a reasonably larger k.

7.4 The Performance of RTS in Adversarial Examples

Data Collection & Baselines. In this section, we further discuss the performance of RTS in adversarial attack scenarios. Adversarial attacks add imperceptible perturbations to images to mislead DNNs. To generate adversarial test inputs, we utilize four state-of-the-art techniques, including FGSM [72], BIM [73], JSMA [74], and CW [75]. These generated adversarial samples are combined with the original test set to form candidate test sets. In addition, we have chosen six DNN&Dataset configurations where the selected DNN models achieve the highest accuracy in their respective datasets. These configurations are as follows: LeNet-5&MNIST, ResNet-20&Fashion, VGG-16&CIFAR, VGG-16&SVHN, ResNet-32&CIFAR-100, and ResNet-20&Caltech-101. The remaining results demonstrate a similar trend and are also available online[6]. We select ATS, LSA, NAC baselines, representing three distinct TS categories (refer to Section 5.3). Considering that among active learning methods, Max$p$ achieves the best results in RQ1 and CLUE has the best model optimization performance in RQ2, we chose Max$p$ as well as CLUE as representatives of active learning.

Evaluation Metrics. Following previous studies [12], we adopt the Average Percentage of Fault-Detection (APFD) metric [76], which can provide overall performance of TS in adversarial examples. A higher APFD value indicates a faster and more efficient detection of misclassifications. Consider a permutation of n test inputs, where k tests are misclassified. Let r_i represent the order of the first test that uncovers the ith misclassified test. The APFD value for this permutation can be calculated using the following formula:

$$APFD = 1 - \frac{\sum_{i=1}^{k} r_i}{kn} + \frac{1}{2n} \tag{12}$$

We normalize the APFD values to the interval $[0, 1]$, ensuring that the TS assigns a higher input priority when the APFD value approaches 1 and a lower selection priority as the APFD value nears 0.

6. https://github.com/swf1996120/RTS

TABLE 11
The performance of RTS in adversarial examples. Values highlighted in red and blue indicate the best and second best.

Baseline	LeNet-5 MNIST	ResNet-20 Fashion	VGG-16 CIFAR	VGG-16 SVHN	ResNet-32 CIFAR-100	ResNet-20 Caltech-101
ATS	0.5811	0.5684	0.5487	0.5743	0.5011	0.5366
LSA	0.6177	0.5766	0.5477	0.5405	0.5127	0.5634
Max$p$	0.5908	0.5778	0.5422	0.5875	0.5210	0.5416
CLUE	0.5069	0.5160	0.4914	0.5123	0.4982	0.5031
NAC	0.4996	0.5000	0.4991	0.4994	0.4991	0.5045
RTS	0.6016	0.5722	0.5680	0.5884	0.5178	0.5490

Results. Table 11 presents the performance of TS methods on the six datasets, with the best results highlighted in red color and the second best highlighted in blue color for each dataset. Among them, NAC and CLUE exhibit the poorest performance, while LSA, Max$p$, and RTS demonstrate advantages on different datasets. RTS achieves the best or second best APFD values in most cases. Specifically, RTS achieves the highest APFD values on VGG-16&CIFAR and VGG-16&SVHN. RTS ranks second best on LeNet-5&MNIST, ResNet-32&CIFAR-100, and ResNet-20&Caltech-101, with a minor difference of 0.0161, 0.0032, and 0.0144 from the best performance, respectively. In RTS, suspect test inputs can be selected by aggregating predictions from multiple similar test inputs. This voting mechanism is designed to alleviate the model's prediction limitations, specifically in cases involving adversarial samples with high confidence levels. RTS aims to achieve consensus among similar inputs, thereby filtering out misleading predictions caused by adversarial perturbations or model vulnerabilities. By aggregating predictions, RTS effectively identifies and prioritizes suspicious test inputs that exhibit inconsistencies or ambiguities in their prediction labels. In conclusion, RTS shows competitive performance compared to other advanced TS methods in adversarial sample scenarios. Future work will focus on investigating RTS's performance on large-scale datasets and models under adversarial attack scenarios.

7.5 Reasons for Improvement of Effectiveness

In the RQ1 and RQ2, we explore the effectiveness of RTS by comparing it to 14 baseline methods. The promising results show that RTS is more effective. This improvement can be attributed to three key aspects, each alleviating a specific issue that has impeded the progress of existing uncertainty-based TS methods.

(1) The presence of noisy data in the candidate test set can significantly impact the failure detection capability of TS methods (as shown in Table 1). However, in practical scenarios, it is challenging for testers to determine the presence and extent of noise. Besides, existing TS methods do not propose strategies to handle test selection scenarios involving noisy data. Therefore, we design an adaptive filtering test noise method by comprehensive evaluating local and global information of image. The results in Table 8 demonstrate that the anti-noise component of RTS effectively identifies noise, with AUC results exceeding 0.70 even for complex datasets like CIFAR. Ablation experiments corroborate that RTS can filter noise to mitigate its detrimental effects and enhance overall effectiveness.

(2) Existing TS methods use hidden layer information or output probabilities to select failure test inputs near the decision boundary. However, DNN models also have strong confidence for some false predictions [19]. Moreover, adversarial input generation techniques, such as C&W [75], aim to generate test inputs that maximize the probability of error classes. These failure-revealing test inputs often lie far from the decision boundary, making existing TS methods less effective or even inapplicable. In contrast, RTS introduces a novel testing perspective by leveraging the class labels of similar instances of test inputs to identify suspicious test cases. Despite the high classification confidence displayed by these failure-revealing test inputs, their class labels are inconsistent with those of similar instances. As a result, RTS can assign a higher test priority to these test inputs. The ablation experiments provide further evidence that the removal of Component 2 results in a varying degree of decrease in the failure detection performance of the RTS.

(3) The probability-tier-matrix-based test metric in RTS offers several advantages over other TS techniques: 1) ***Consideration of both test input uncertainty and diversity***: The proposed test metric in RTS can utilize both uncertainty and diversity aspects when selecting test inputs. The selected inputs can cover a wide range of scenarios and capture different aspects of the model's behavior. 2) ***Discretization of prediction probabilities***: By leveraging the probability tier matrix, the discretization process serves two primary functions: it discerns and highlights significant variations in output probabilities and also standardizes these probabilities inherently. In real-world scenarios, training datasets may exhibit imbalances across categories, resulting in varied sample sizes among them. Such discrepancies induce models to be overly confident in well-represented classes, while showing reduced confidence in sparsely represented ones. However, the inter-class output probability differences can be counterproductive, especially during cross-class comparisons, leading the TS method to habitually lean towards the lesser-represented classes. Through discretization, we transition from potentially biased raw probabilities to standardized intervals, ensuring a consistent and equitable comparison of the model's predictions across all categories. 3) ***Class-level combination coverage***: RTS introduces the concept of class-level combination coverage based on the probability tier matrix. By exploring different output probability combinations in the selected test inputs, RTS can realize the complexity and richness of the test output. Despite the state-of-the-art ATS proposes the concept of *fault pattern* to achieve diversity in test inputs, but compared to RTS, ATS has the following limitations: 1) ***Limited applicability***: ATS is designed specifically for classification tasks with class labels greater than 3; 2) ***Computational complexity***: for m-class classification tasks, ATS needs to compute two-by-two combinations among $m-1$ classes to obtain the *fault pattern* in each sub-output space. For complex datasets, such as ImageNet, the computational complexity is significant.

7.6 Limitations of RTS

7.6.1 Similarity Assumption

Component 2 is proposed based on an intuition that class labels of instances similar to the test input can guide the identification of suspicious test inputs. Admittedly, this assumption may not hold true in all cases within image classification and computer vision tasks, especially in entangled datasets where deep learning models may struggle to recognize recurring data patterns. As a result, achieving class label consistency for similar instances may not always be possible, resulting in some successful test cases being classified in the ***SPS***. Nevertheless, we propose Component 3 to prioritize the inputs in the ***SPS***, trying to rank successful test inputs towards the end. Moreover, in all 48 experimental configurations, we demonstrate that Component 2, which is based on the intuition mentioned above, can enhance the failure detection performance in most cases. Furthermore, in this paper, we would like to convey the message that making use of the relationships between test inputs can further complement existing test criteria and derive new test selection methods. Our evaluation is performed on a relatively low-dimensional dataset, and we hope that our work can be used as a starting point for test selection in higher dimensional and more complex datasets.

7.6.2 SSIM-based Noise Filtering Component

While the SSIM-based noise filtering component (i.e., Component 1) demonstrates effectiveness in reducing noise (refer to Table 8), it falls short of achieving metrics surpassing the 90% threshold overall. This limitation indeed impacts RTS's failure detection performance to a certain extent. From Table 9, RTS achieves the fifth-best failure detection performance among 15 methods on noiseless candidate sets, regardless of the selection rate. One plausible explanation for RTS not achieving peak failure detection on noiseless candidate sets is the SSIM-based noise filtering component incorrectly classifies valid test inputs as noise, potentially containing some failure-revealing test inputs. However, despite this, RTS still manages to attain significant performance improvements on the noisy candidate sets (Tables 5 and 6), demonstrating that Component 1 effectively enhances the robustness of RTS in candidate sets with noise. Given the benefits of RTS on noisy sets and achieving the best or second-best model optimization results on noiseless candidate sets, we maintain our recommendation for RTS as the preferred test selection method based on the experimental results, regardless of whether the candidate set contains noise or not. Furthermore, we recognize that developing more advanced and comprehensive noise filtering algorithms holds promise for enhancing the effectiveness of RTS. This avenue will be a central focus of our future research efforts.

Moreover, we employ the p-hash and SSIM algorithms to evaluate image similarity. In fact, while both p-hash and SSIM are sensitive to some transformations, these algorithms have different sensitivities. The p-hash algorithm can find similar local structures and demonstrates robustness against transformations like lighting and rotation by leveraging the local information of an image. While SSIM is more suitable for capturing global structural similarities. By sequentially integrating these algorithms, we can assess image similarity in a wider range of application scenarios, covering both local and global information. However, we acknowledge the challenge of comprehensively capturing all types of transitions using just p-hash and SSIM. This aspect will be

a focus of our future work, as we endeavor to explore more comprehensive guidelines for assessing test input similarity.

7.6.3 RTS's Application

As explained in Section 4.1, our approach is primarily designed and evaluated for image classification models. The current form of RTS may not be directly applicable to regression models due to the distinct objectives and evaluation metrics employed in classification and regression tasks. Classification models aim to categorize data into discrete classes, while regression models focus on predicting continuous values or estimating variable relationships. The specific techniques and strategies utilized in our method, such as the probability tier matrix for quantifying test input diversity, may require adaptation or replacement to align with the requirements of regression tasks. Future research should explore and develop specialized methods and techniques tailored to the unique characteristics of regression tasks, such as utilizing model interpretability approaches.

8 Threats to Validity

In this section, we examine potential threats to our paper's validity.

8.1 Threats to Validity

Threats to internal validity pertain to factors within our study that have the potential to impact our findings.

Similarity metric selection. This paper adopts SSIM as the similarity metric to filter out the noisy data since SSIM can provide acceptable visual agreement and computational overhead. However, SSIM is an image quality metric. Currently, RTS cannot be directly applied to non-image type datasets, such as text, video, and audio. But we believe that RTS can be easily migrated to non-image classification tasks by changing the way we calculate the similarity. For example, for text-type test inputs, we can obtain each test input's feature representation by large-scale pre-training models and use cosine distance as a similarity criterion.

Parameters settings. For hyperparameters, k is important because it determines how test inputs are identified as successful or suspicious and impact the performance in identifying noise. Large k value can improve the precision of identifying noise and suspicious test inputs but lead to high computation overhead. We experimentally find that $k = 5$ can have satisfactory cost-effective performances. Regarding randomness, we repeat 3 times per experiment to ensure reliable results. In our study, the choice to introduce 20% pollution to simulate real-world situations raises a potential threat to the validity of our results. The introduction of 20% pollution is a design decision (that follows previous work [15]) to strike a balance between disrupting the data and maintaining its essential characteristics. To mitigate the threat and improve the validity of our study, we conduct experiments on different datasets and models, and the results all show the superiority of RTS. In future research, we will consider exploring different levels of pollution, evaluating the approach on diverse datasets with varying characteristics, and conducting sensitivity analyses to assess the robustness of the RTS to different pollution thresholds.

8.2 Threats to External Validity

External validity pertains specifically to the extent to which our experimental results can be generalized.

Test subject selection. The main threats to validity come from the datasets and DNN models. We can not claim that the dataset and DNN models used in this work can be representative of all datasets and DNN models. However, the adopted datasets and DNN models are well-known and typical, and are widely-used in test-selection-related research. Besides, for each dataset, we utilize two DNN models with different architectures to evaluate the performance of RTS. Following the previous works [9], [10], [12], [15], we focus on DNN classification tasks, and generative tasks will be one of our future works.

Image transformation. In our experiments, we utilize seven data augmentation operators to generate additional test inputs and construct the candidate test sets. However, this would introduce the potential threat of label inconsistency between the transformed images and the original images. For example, the rotation operation may cause the number 6 (9) to become 9 (6) in the MNIST dataset. To mitigate this threat, for the configurable parameters of the image transformation techniques, we strictly follow the authors' suggested settings or adopt the default settings [10], [15], [57]. Furthermore, in order to ensure semantic consistency, we employ three volunteers to conduct a thorough check of the transformed images. By implementing these precautions, we can minimize the risk of label inconsistency and enhance the reliability of our experimental results.

Noise data construction. Other threats to validity arise from the fact that existing publicly available image datasets are meticulously labeled and collected, typically devoid of noisy or invalid data. In order to investigate the performance of TS methods in real-world scenarios involving the presence of invalid data, we have to design several data contamination scenarios. These scenarios are chosen to be diverse, encompass four different types, and have been widely adopted in previous studies [15]. However, it is important to acknowledge that these artificially constructed contamination scenarios may not fully capture the complexity and variability of real-world noisy data. The performance of TS methods on such real noisy datasets remains an area for future exploration. It is crucial to dedicate efforts to constructing and collecting datasets that contain invalid data in realistic scenarios. This will enable us to further investigate the performance of RTS and other TS methods in real-world scenarios.

9 Related works

This section introduces the related works on two aspects of test selection: test selection for DNNs and test selection in traditional software.

9.1 Test Selection for DNNs

In the past few years, many test selection metrics have been proposed to reduce the labeling effort. On the one hand, Li et al. [61] view the idea behind traditional structural coverage as conditioning for variance reduction. Inspired by this insight, they propose a stratified sampling method CSS based

on the model prediction confidence and a cross-entropy-based sampling method CSE. After that, Zhou et al. [77] propose DeepReduce based on a two-phase strategy. First, DeepReduce selects test inputs satisfying testing adequacy. Then, more test inputs would be selected to approximate the distribution between the entire test set and the selected set by leveraging relative entropy minimization. Recently, Chen et al. [68] propose a new accuracy estimation method, namely PACE. PACE first uses cluster analysis to divide the test inputs into different groups. Then, PACE uses the MMD-critic algorithm [78] to select prototypes from each group according to the group size. Also, PACE draws on the idea of adaptive random testing to choose test cases from minority spaces to achieve test case diversity. The above-mentioned works focus on selecting data that can be used to estimate model performance and represent the entire set. As the chosen test set is usually small-scale and achieves comparable evaluation performance to the original test set, it can significantly reduce the overhead of labeling and save model evaluation time. In this paper, we use CES, which is representative among the above-mentioned test selection methods, as our benchmark. Experimental results show that such type of test selection cannot effectively identify failure-revealing test inputs and have limited model optimization performances. In contrast, RTS aims to select data more likely to be incorrectly classified by the model and then retrain a better model with the selected data.

Additionally, Kim et al. [16] utilize the similarity between the training set and test inputs to propose the surprise-guided testing metrics, which measure how surprising or out-of-distribution the DNN activations for a given test input are compared to the activation observed on the training set. The surprise-based TS methods select test inputs as close to the DNN decision boundary as possible. As discussed in Section 3.2, such methods cannot select failure-revealing test inputs that far away from the decision boundary of DNN. Compared to these surprise-based TS methods, RTS further considers the relationships between test inputs and similar instances for better selection and experimental results show RTS achieves better failure-revealing capability. For the adversarial robustness of DNNs, Wang et al. [79] propose robustness-oriented testing metrics as well as selection metrics. However, their metric can only be applied to adversarial attacks. Wang et al. [19] propose a new selection approach via mutation analysis. They use both image mutations and DNN model mutations to select test inputs that are likely to reveal DNN bugs, and hence require the inner information of the model. However, releasing the inner information of the DNN model may lead to serious consequences, such as model inversion attacks [79]. In contrast, RTS is black-box TS method and achieves good failure detection performance using only the training samples and the output probabilities output by the model. Thus, RTS is considerably more practical. Recently, Gao et al. [15] propose a novel adaptive test selection method (ATS) that is useful to guide model retraining and achieve promising results. Compared with ATS, RTS can better handle noise in candidate test inputs and achieve better performance. Besides, RTS is suitable for any classification task, whereas ATS cannot select test inputs for binary classification models.

9.2 Test Selection for Traditional Software

In traditional software testing, test selection refers to selecting and executing test cases that are affected by software changes in regression testing since the execution results of the test cases that are not affected by code changes should not change [80]. Therefore, test selection is not only temporary (i.e., selection result changes with the evolution of the project) but also is modification-aware (i.e., concerns the modified parts of the program), typically involving a white-box static analysis.

To this end, various approaches have been proposed using different criteria. According to the descriptions of Yoo and Harman [76], the test selection can divided into following parts: Integer Programming [81], data flow analysis [82], symbolic execution [83], CFG graph-walking [84], dynamic slicing [85], textual difference in source code [86], path analysis [87], SDG slicing [88], modification detection [89], firewall [90], CFG cluster identification [91] and design-based testing [92]. Later, For Java projects, Gligoric et al. [93] propose a file-level dynamic test selection method through the changes of bytecode class files. Legunsen et al. [94] evaluate and compare different static test selection methods on 986 revisions of 22 open-source projects. Zhang [95] combine the strengths of existing dynamic test selection methods at different granularities to propose the first hybrid approach. Guizzo et al. [96] use the regression test selection techniques as a core component of the GI (i.e., genetic improvement, an artificial intelligence technique improving a given property of an existing software [97]) search process and demonstrate test selection can speeds up the whole GI process.

10 Conclusion and Future Work

In this paper, we propose RTS, a robust test selection for deep neural networks. Considering noisy data in real scenarios, RTS combines retrieval and structural similarity index to filter noisy data. Besides, RTS utilizes class information of similar instances of test inputs and majority voting to effectively distinguish suspicious and successful test inputs. Subsequently, we introduce probability tier matrix and design a novel test metric. We propose a fitness function to determine which test in the candidate set is more suitable to be labeled. The experimental results demonstrate that RTS can identify more failure-revealing test inputs within a limited selection size. Additionally, the selected test inputs can be utilized to retrain the model to improve the quality of the model. Different from current test selection methods using prediction probability or neuron activation state, RTS explores the use of relationships between test inputs and their similar instances to identify failure-revealing test inputs, which provides a new perspective for subsequent TS methods. We expect that RTS can inspire testers to propose more robust TS methods to deal with more complex test scenarios. Limited by our noise identification method, RTS may be more suitable for images. In the future, we will propose a general noise identification method to expand the kind of data that RTS can be applied to. Following the previous works [9], [10], [12], [15], we focus on DNN classification tasks, and generative tasks will be one of our future works.

11 Acknowledgements

We appreciate the insightful insights provided by anonymous reviewers to improve the quality of the paper. This work was supported in part by the National Key Research and Development Project (No. 2021YFB1714200), the National Natural Science Foundation of China (No. 62372071), the Fundamental Research Funds for the Central Universities (No. 2022CDJDX-005), the Chongqing Technology Innovation and Application Development Project (No. CSTB2022TIADSTX0007 and No. CSTB2022TIAD-KPX0067).

References

[1] S. Gao, C. Gao, Y. He, J. Zeng, L. Y. Nie, and X. Xia, "Code structure guided transformer for source code summarization," *CoRR*, vol. abs/2104.09340, 2021. [Online]. Available: https://arxiv.org/abs/2104.09340

[2] C. Zeng, Y. Yu, S. Li, X. Xia, Z. Wang, M. Geng, B. Xiao, W. Dong, and X. Liao, "degraphcs: Embedding variable-based flow graph for neural code search," *CoRR*, vol. abs/2103.13020, 2021. [Online]. Available: https://arxiv.org/abs/2103.13020

[3] Y. Sun, D. Liang, X. Wang, and X. Tang, "Deepid3: Face recognition with very deep neural networks," *CoRR*, vol. abs/1502.00873, 2015. [Online]. Available: http://arxiv.org/abs/1502.00873

[4] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," *CoRR*, vol. abs/1604.07316, 2016.

[5] D. Wakabayashi, "Self-driving uber car kills pedestrian in arizona, where robots roam," *The New York Times*, vol. 19, no. 03, 2018.

[6] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should I trust you?": Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, B. Krishnapuram, M. Shah, A. J. Smola, C. C. Aggarwal, D. Shen, and R. Rastogi, Eds. ACM, 2016, pp. 1135–1144. [Online]. Available: https://doi.org/10.1145/2939672.2939778

[7] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*. IEEE Computer Society, 2009, pp. 248–255.

[8] A. W. Senior, R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T. Green, C. Qin, A. Zídek, A. W. R. Nelson, A. Bridgland, H. Penedones, S. Petersen, K. Simonyan, S. Crossan, P. Kohli, D. T. Jones, D. Silver, K. Kavukcuoglu, and D. Hassabis, "Improved protein structure prediction using potentials from deep learning," *Nat.*, vol. 577, no. 7792, pp. 706–710, 2020. [Online]. Available: https://doi.org/10.1038/s41586-019-1923-7

[9] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: automated whitebox testing of deep learning systems," *Commun. ACM*, vol. 62, no. 11, pp. 137–145, 2019. [Online]. Available: https://doi.org/10.1145/3361566

[10] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepgauge: multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. ACM, 2018, pp. 120–131. [Online]. Available: https://doi.org/10.1145/3238147.3238202

[11] 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, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 2018, pp. 303–314. [Online]. Available: https://doi.org/10.1145/3180155.3180220

[12] 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 *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*. ACM, 2020, pp. 177–188.

[13] T. Byun, V. Sharma, A. Vijayakumar, S. Rayadurgam, and D. D. Cofer, "Input prioritization for testing neural networks," in *IEEE International Conference On Artificial Intelligence Testing, AITest 2019, Newark, CA, USA, April 4-9, 2019*. IEEE, 2019, pp. 63–70. [Online]. Available: https://doi.org/10.1109/AITest.2019.000-6

[14] L. Zhang, X. Sun, Y. Li, and Z. Zhang, "A noise-sensitivity-analysis-based test prioritization technique for deep neural networks," *arXiv preprint arXiv:1901.00054*, 2019.

[15] X. Gao, Y. Feng, Y. Yin, Z. Liu, Z. Chen, and B. Xu, "Adaptive test selection for deep neural networks," in *Proceedings of the 44th International Conference on Software Engineering, ICSE 2022*, 2022.

[16] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 2019, pp. 1039–1049. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00108

[17] G. Jahangirova and P. Tonella, "An empirical evaluation of mutation operators for deep learning systems," in *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 2020, pp. 74–84. [Online]. Available: https://doi.org/10.1109/ICST46399.2020.00018

[18] M. Weiss and P. Tonella, "Simple techniques work surprisingly well for neural network test prioritization and active learning (replicability study)," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*. ACM, 2022, pp. 139–150. [Online]. Available: https://doi.org/10.1145/3533767.3534375

[19] Z. Wang, H. You, J. Chen, Y. Zhang, X. Dong, and W. Zhang, "Prioritizing test inputs for deep neural networks via mutation analysis," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 397–409. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00046

[20] W. Zhang, Y. Kinoshita, and H. Kiya, "Image-enhancement-based data augmentation for improving deep learning in image classification problem," in *IEEE International Conference on Consumer Electronics - Taiwan, ICCE-TW 2020, Taoyuan, Taiwan, September 28-30, 2020*. IEEE, 2020, pp. 1–2. [Online]. Available: https://doi.org/10.1109/ICCE-Taiwan49838.2020.9258292

[21] L. Ertöz, M. Steinbach, and V. Kumar, "Finding clusters of different sizes, shapes, and densities in noisy, high dimensional data," in *Proceedings of the 2003 SIAM international conference on data mining*. SIAM, 2003, pp. 47–58.

[22] Y. Yang, "Noise reduction in a statistical approach to text categorization," in *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, 1995, pp. 256–263.

[23] L. Yi, B. Liu, and X. Li, "Eliminating noisy information in web pages for data mining," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003, pp. 296–305.

[24] J. Anaya and A. Barbu, "RENOIR - A benchmark dataset for real noise reduction evaluation," *CoRR*, vol. abs/1409.8230, 2014. [Online]. Available: http://arxiv.org/abs/1409.8230

[25] H. Xiong, G. Pandey, M. Steinbach, and V. Kumar, "Enhancing data analysis with noise removal," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 3, pp. 304–319, 2006.

[26] T. Tuor, S. Wang, B. J. Ko, C. Liu, and K. K. Leung, "Overcoming noisy and irrelevant data in federated learning," in *25th International Conference on Pattern Recognition, ICPR 2020, Virtual Event / Milan, Italy, January 10-15, 2021*. IEEE, 2020, pp. 5020–5027. [Online]. Available: https://doi.org/10.1109/ICPR48806.2021.9412599

[27] O. Sagi and L. Rokach, "Ensemble learning: A survey," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 4, p. e1249, 2018.

[28] J. S. Denker and Y. LeCun, "Transforming neural-net output levels to probability distributions," in *Advances in Neural Information Processing Systems 3, [NIPS Conference, Denver, Colorado, USA, November 26-29, 1990]*. Morgan Kaufmann, 1990, pp. 853–859.

[29] X. Xie, J. W. K. Ho, C. Murphy, G. E. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *J. Syst. Softw.*, vol. 84, no. 4, pp. 544–558, 2011. [Online]. Available: https://doi.org/10.1016/j.jss.2010.11.920

[30] X. Du, X. Xie, Y. Li, L. Ma, Y. Liu, and J. Zhao, "Deepstellar: model-based quantitative analysis of stateful deep learning systems," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 2019, pp. 477–487. [Online]. Available: https://doi.org/10.1145/3338906.3338954

[31] D. Cheng, C. Cao, C. Xu, and X. Ma, "Manifesting bugs in machine learning code: An explorative study with mutation

testing," in *2018 IEEE International Conference on Software Quality, Reliability and Security, QRS 2018, Lisbon, Portugal, July 16-20, 2018*. IEEE, 2018, pp. 313–324. [Online]. Available: https://doi.org/10.1109/QRS.2018.00044

[32] A. Aggarwal, P. Lohia, S. Nagar, K. Dey, and D. Saha, "Black box fairness testing of machine learning models," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 2019, pp. 625–635. [Online]. Available: https://doi.org/10.1145/3338906.3338937

[33] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "Dlfuzz: differential fuzzing testing of deep learning systems," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 2018, pp. 739–743. [Online]. Available: https://doi.org/10.1145/3236024.3264835

[34] S. Udeshi and S. Chattopadhyay, "Grammar based directed testing of machine learning systems," *IEEE Trans. Software Eng.*, vol. 47, no. 11, pp. 2487–2503, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2019.2953066

[35] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Trans. Software Eng.*, vol. 48, no. 2, pp. 1–36, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2019.2962027

[36] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[37] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[38] F. Wilcoxon, *Individual comparisons by ranking methods*. Springer, 1992.

[39] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, and X. Xia, "A survey on adaptive random testing," *IEEE Trans. Software Eng.*, vol. 47, no. 10, pp. 2052–2083, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2019.2942921

[40] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal, "Searching for cognitively diverse tests: Towards universal test diversity metrics," in *First International Conference on Software Testing Verification and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008, Workshops Proceedings*. IEEE Computer Society, 2008, pp. 178–186. [Online]. Available: https://doi.org/10.1109/ICSTW.2008.36

[41] N. Alshahwan and M. Harman, "Augmenting test suites effectiveness by increasing output diversity," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE Computer Society, 2012, pp. 1345–1348. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227083

[42] ——, "Coverage and fault detection of the output-uniqueness test selection criteria," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*. ACM, 2014, pp. 181–192. [Online]. Available: https://doi.org/10.1145/2610384.2610413

[43] H. D. Menéndez, M. Boreale, D. Gorla, and D. Clark, "Output sampling for output diversity in automatic unit test generation," *IEEE Trans. Software Eng.*, vol. 48, no. 2, pp. 295–308, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2020.2987377

[44] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Trans. Image Process.*, vol. 13, no. 4, pp. 600–612, 2004. [Online]. Available: https://doi.org/10.1109/TIP.2003.819861

[45] E. Klinger and D. Starkweather, "phash: The open source perceptual hash library," *https://www.phash.org/docs/*, 2021.

[46] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, 2017. [Online]. Available: http://doi.acm.org/10.1145/3065386

[47] L. Deng, "The MNIST database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 141–142, 2012. [Online]. Available: https://doi.org/10.1109/MSP.2012.2211477

[48] D. D. Lewis and W. A. Gale, "A sequential algorithm for training text classifiers," in *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval. Dublin, Ireland, 3-6 July 1994 (Special Issue of the SIGIR Forum)*. ACM/Springer, 1994, pp. 3–12. [Online]. Available: https://doi.org/10.1007/978-1-4471-2099-5_1

[49] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[50] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar10." *https://www.cs.toronto.edu/ kriz/cifar.html*, 2009.

[51] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," 2011.

[52] L. Fei-Fei, R. Fergus, and P. Perona, "Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories," in *2004 conference on computer vision and pattern recognition workshop*. IEEE, 2004, pp. 178–178.

[53] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[54] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[55] 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.

[56] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *J. Big Data*, vol. 6, p. 60, 2019. [Online]. Available: https://doi.org/10.1186/s40537-019-0197-0

[57] Z. Liu, Y. Feng, Y. Yin, and Z. Chen, "Deepstate: selecting test suites to enhance the robustness of recurrent neural networks," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 598–609.

[58] W. Ma, M. Papadakis, A. Tsakmalis, M. Cordy, and Y. L. Traon, "Test selection for deep learning systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, pp. 13:1–13:22, 2021. [Online]. Available: https://doi.org/10.1145/3417330

[59] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, ser. JMLR Workshop and Conference Proceedings, M. Balcan and K. Q. Weinberger, Eds., vol. 48. JMLR.org, 2016, pp. 1050–1059. [Online]. Available: http://proceedings.mlr.press/v48/gal16.html

[60] V. Prabhu, A. Chandrasekaran, K. Saenko, and J. Hoffman, "Active domain adaptation via clustering uncertainty-weighted embeddings," in *2021 IEEE/CVF International Conference on Computer Vision, ICCV 2021, Montreal, QC, Canada, October 10-17, 2021*. IEEE, 2021, pp. 8485–8494. [Online]. Available: https://doi.org/10.1109/ICCV48922.2021.00839

[61] Z. Li, X. Ma, C. Xu, C. Cao, J. Xu, and J. Lü, "Boosting operational DNN testing efficiency through conditioning," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, 2019, pp. 499–509. [Online]. Available: https://doi.org/10.1145/3338906.3338930

[62] D. Hendrycks and K. Gimpel, "A baseline for detecting misclassified and out-of-distribution examples in neural networks," *arXiv preprint arXiv:1610.02136*, 2016.

[63] J. Cuzick, "A wilcoxon-type test for trend," *Statistics in medicine*, vol. 4, no. 1, pp. 87–90, 1985.

[64] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[65] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[66] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson, "Estimating the support of a high-dimensional distribution," *Neural Comput.*, vol. 13, no. 7, pp. 1443–1471, 2001. [Online]. Available: https://doi.org/10.1162/089976601750264965

[67] R. J. G. B. Campello, D. Moulavi, A. Zimek, and J. Sander, "Hierarchical density estimates for data clustering, visualization, and outlier detection," *ACM Trans. Knowl. Discov. Data*, vol. 10, no. 1, pp. 5:1–5:51, 2015. [Online]. Available: https://doi.org/10.1145/2733381

[68] J. Chen, Z. Wu, Z. Wang, H. You, L. Zhang, and M. Yan, "Practical accuracy estimation for efficient deep neural network testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, pp. 30:1–30:35, 2020. [Online]. Available: https://doi.org/10.1145/3394112

[69] A. P. Bradley, "The use of the area under the ROC curve in the evaluation of machine learning algorithms," *Pattern*

Recognit., vol. 30, no. 7, pp. 1145–1159, 1997. [Online]. Available: https://doi.org/10.1016/S0031-3203(96)00142-2

[70] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Software Eng.*, vol. 34, no. 4, pp. 485–496, 2008. [Online]. Available: https://doi.org/10.1109/TSE.2008.35

[71] J. Nam and S. Kim, "CLAMI: defect prediction on unlabeled datasets (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE Computer Society, 2015, pp. 452–463. [Online]. Available: https://doi.org/10.1109/ASE.2015.56

[72] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. [Online]. Available: http://arxiv.org/abs/1412.6572

[73] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=HJGU3Rodl

[74] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. IEEE, 2016, pp. 372–387. [Online]. Available: https://doi.org/10.1109/EuroSP.2016.36

[75] N. Carlini and D. A. Wagner, "Towards evaluating the robustness of neural networks," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 39–57. [Online]. Available: https://doi.org/10.1109/SP.2017.49

[76] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Softw. Test. Verification Reliab.*, vol. 22, no. 2, pp. 67–120, 2012. [Online]. Available: https://doi.org/10.1002/stv.430

[77] J. Zhou, F. Li, J. Dong, H. Zhang, and D. Hao, "Cost-effective testing of a deep learning model through input reduction," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 289–300.

[78] B. Kim, O. Koyejo, and R. Khanna, "Examples are not enough, learn to criticize! criticism for interpretability," in *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, 2016, pp. 2280–2288. [Online]. Available: https://proceedings.neurips.cc/paper/2016/hash/5680522b8e2bb01943234bce7bf84534-Abstract.html

[79] J. Wang, J. Chen, Y. Sun, X. Ma, D. Wang, J. Sun, and P. Cheng, "Robot: Robustness-oriented testing for deep learning systems," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 300–311. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00038

[80] O. Legunsen, A. Shi, and D. Marinov, "STARTS: static regression test selection," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. IEEE Computer Society, 2017, pp. 949–954. [Online]. Available: https://doi.org/10.1109/ASE.2017.8115710

[81] K. F. Fischer, "A test case selection method for the validation of software maintenance modifications," 1977.

[82] M. J. Harrold and M. Souffa, "An incremental approach to unit testing during maintenance," in *1988 Conference on Software Maintenance*. IEEE Computer Society, 1988, pp. 362–367.

[83] S. S. Yau and Z. Kishimoto, "Method for revalidating modified programs in the maintenance phase." in *Proceedings-IEEE Computer Society's International Computer Software & Applications Conference*. IEEE, 1987, pp. 272–277.

[84] G. Rothermel and M. J. Harrold, "A safe, efficient algorithm for regression test selection," in *1993 Conference on Software Maintenance*. IEEE, 1993, pp. 358–367.

[85] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London, "Incremental regression testing," in *1993 Conference on Software Maintenance*. IEEE, 1993, pp. 348–357.

[86] F. I. Vokolos and P. G. Frankl, "Empirical evaluation of the textual differencing regression testing technique," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 1998, pp. 44–53.

[87] P. Benedusi, A. Cmitile, and U. De Carlini, "Post-maintenance testing based on path change analysis," in *1988 Conference on Software Maintenance*. IEEE Computer Society, 1988, pp. 352–361.

[88] S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," in *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1993, pp. 384–396.

[89] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo, "Testtube: A system for selective regression testing," in *Proceedings of 16th International Conference on Software Engineering*. IEEE, 1994, pp. 211–220.

[90] L. J. White, V. Narayanswamy, T. Friedman, M. Kirschenbaum, P. Piwowarski, and M. Oha, "Test manager: A regression testing tool," in *1993 Conference on Software Maintenance*. IEEE, 1993, pp. 338–347.

[91] J. Laski and W. Szermer, "Identification of program modifications and its applications in software maintenance," in *Proceedings Conference on Software Maintenance 1992*. IEEE Computer Society, 1992, pp. 282–283.

[92] L. C. Briand, Y. Labiche, and S. He, "Automating regression test selection based on uml designs," *Information and Software Technology*, vol. 51, no. 1, pp. 16–30, 2009.

[93] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*. ACM, 2015, pp. 211–222. [Online]. Available: https://doi.org/10.1145/2771783.2771784

[94] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. ACM, 2016, pp. 583–594. [Online]. Available: https://doi.org/10.1145/2950290.2950361

[95] L. Zhang, "Hybrid regression test selection," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 2018, pp. 199–209. [Online]. Available: https://doi.org/10.1145/3180155.3180198

[96] G. Guizzo, J. Petke, F. Sarro, and M. Harman, "Enhancing genetic improvement of software with regression test selection," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 1323–1333. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00120

[97] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic improvement of software: A comprehensive survey," *IEEE Trans. Evol. Comput.*, vol. 22, no. 3, pp. 415–432, 2018. [Online]. Available: https://doi.org/10.1109/TEVC.2017.2693219