

Enhancing Fault Detection for Large Language Models via Mutation-Based Confidence Smoothing

QIANG HU*, The University of Tokyo, Japan
 JIN WEN*, University of Luxembourg, Luxembourg
 MAXIME CORDY, University of Luxembourg, Luxembourg
 YUHENG HUANG, The University of Tokyo, Japan
 XIAOFEI XIE, Singapore Management University, Singapore
 LEI MA, The University of Tokyo & University of Alberta, Japan & Canada

Large language models (LLMs) achieved great success in multiple application domains and attracted huge attention from different research communities recently. Unfortunately, even for the best LLM, there still exist many *faults* that LLM cannot correctly predict. Such faults will harm the usability of LLMs. How to quickly reveal them in LLMs is important, but challenging. The reasons are twofold, 1) the heavy labeling effort for preparing the test data, and 2) accessing closed-source LLMs such as GPT4 is money-required. To handle this problem, in the traditional deep learning testing field, test selection methods have been proposed for efficiently testing deep learning models by prioritizing faults. However, the usefulness of these methods on LLMs is unclear and under exploration. In this paper, we first study the effectiveness of existing fault detection methods for LLMs. Experimental results on four different tasks (including both code tasks and natural language processing tasks) and four LLMs (e.g., LLaMA and GPT4) demonstrated that existing fault detection methods cannot perform well on LLMs (e.g., seven out of eight methods perform worse than random selection on LLaMA). To enhance existing fault detection methods, we propose **MuCS**, a prompt **M**utation-based prediction **C**onfidence **S**moothing method for LLMs. Concretely, we mutate the prompts and compute the average prediction confidence of all mutants as the input of fault detection methods. The results show that our proposed solution significantly enhances existing methods with the improvement of test relative coverage by up to 97.64%.

ACM Reference Format:

Qiang Hu*, Jin Wen*, Maxime Cordy, Yuheng Huang, Xiaofei Xie, and Lei Ma. 2024. Enhancing Fault Detection for Large Language Models via Mutation-Based Confidence Smoothing. 1, 1 (April 2024), 21 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION

Since the last few years, the potential of large language models (LLMs) has brought hope to general artificial intelligence (AGI). LLMs achieved promising and even the best results compared to other methods in many areas, such as question answering [40], sentiment analysis [37], and code understanding [22] that make LLMs become the first choice of deep learning models in

* Equal Contribution.

Authors' addresses: Qiang Hu*, The University of Tokyo, Japan; Jin Wen*, University of Luxembourg, Luxembourg; Maxime Cordy, University of Luxembourg, Luxembourg; Yuheng Huang, The University of Tokyo, Japan; Xiaofei Xie, Singapore Management University, Singapore; Lei Ma, The University of Tokyo & University of Alberta, Japan & Canada.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM XXXX-XXXX/2024/4-ART

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

such tasks. Besides the most famous application ChatGPT¹, multiple LLMs have been proposed, e.g., LLaMA [31], Alpaca [30], and Cerebras-GPT [5]. More interestingly, focusing on software engineering (SE) tasks, a series of code-related LLMs with good programming ability have been released to help the software development such as StarCoder [18] and CodeLLaMA [28]. Learning to use LLMs is already a basic skill for us in daily life.

Despite the success, the same as other types of deep learning models, LLMs also suffer from different problems that limit their practical usage. For example, LLMs sometimes generate unrelated outputs to the inputs or incorrect results misaligned with established world knowledge, which is known as the hallucination problem [38]. Besides, recent work [6] showed that LLMs can be easily fooled by adversarial attacks and are not robust. Furthermore, considering SE tasks, existing code-related LLMs cannot capture equivalent semantics when the given natural language prompt expresses the same meaning in different languages [26]. Those limitations remind us it is necessary to comprehensively evaluate and test the capabilities of LLMs before deployment.

The basic way to test LLMs is to prepare as much as possible test data to assess the performance of LLMs accordingly. However, preparing such test data is heavy work due to the complicated process of data collection, data cleaning, and data labeling. Especially, data labeling requires human effort with domain knowledge which is the most challenging part. Moreover, some LLMs are closed-source which requires funding expenses, e.g., it costs \$0.03/1K tokens of inputs for GPT4 using OpenAI API. Therefore, it is almost impossible to test LLMs for people who have limited budgets. How to efficiently test the performance of LLMs with less effort becomes an urgent problem.

To tackle the data labeling issue, in the field of deep learning testing, multiple fault detection methods that quickly identify fault data (data the model has incorrect prediction) in the test set without using the labeling information [14] have been proposed. Based on the information required, existing methods can be divided into learning-based methods and output-based methods. Those methods have been widely studied in the classical deep learning models and have proven to be effective in saving the labeling budget during testing. However, there are no works that explore the usefulness of fault detection methods for LLMs. It is unclear whether we can directly employ such methods to test LLMs and save our effort or not.

To bridge this gap, in this paper, we first study the effectiveness of existing fault detection methods on LLMs. The major challenge in conducting such a study is that most existing fault detection methods are proposed for classification tasks and use the output probabilities for the selection. It is necessary to design the prompt to guide LLMs to produce the output with their confidence. To solve this, for datasets that have more than two classes, we add examples in each class in the prompt as guidance to ask LLMs to predict new data. In total, our study covers nine fault detection methods including both learning-based methods, e.g., TestRank [19], and output-based methods, e.g., ATS [11]. Based on the experiment results collected from four datasets (e.g., sentiment analysis and code clone detection) and four LLMs (e.g., LLaMA and GPT4), we found that 1) GPT models are more confident in their outputs (e.g., 0.84 vs. 0.76 of GPT3.5 vs. CodeLLaMA), 2) LLMs are not well-calibrated and overconfident in clone detection, problem classification, and news classification tasks, and 3) existing fault detection methods perform poorly on LLMs compared to their performance on classical deep learning models. For example, seven out of eight methods perform worse than random selection on LLaMA. There is a need to propose methods to further enhance existing fault detection methods.

To do so, inspired by [20, 33] which utilize mutation testing techniques to help test deep learning models, we propose **MuCS**, a prompt **M**utation-based prediction **C**onfidence **S**moother method to enhance existing fault detection methods. Specifically, we mutate the prompt by transformation

¹OpenAI, ChatGPT, <https://chat.openai.com/>

methods to generate a set of prompt mutants first. Here, both text augmentation methods and code refactoring methods have been considered in MuCS. Then, we collect the output probabilities of all mutants and compute the average prediction confidence. Finally, we use the averaged confidence to perform fault detection using current fault detection methods. We compare the effectiveness of fault detection methods with and without our confidence smoothing method and found that using MuCS significantly enhances the performance of existing methods by up to 97.62%.

To summarize, the main contributions of this paper are:

- This is the first study that explores the effectiveness of fault detection methods on LLMs. We found that 1) existing methods that perform well on traditional deep learning models cannot correctly detect faults in LLMs; 2) LLMs are overconfident in clone detection, problem classification, and news classification tasks; 3) the prediction confidence of LLMs is concentrated in a few intervals.
- We propose MuCS, a prompt mutation-based method to smooth the confidence of LLMs and enhance the effectiveness of existing fault detection methods.

The rest of this paper is organized as follows. Section 2 reviews the related works. Section 3 presents the design of our empirical study and Section 4 summarizes the empirical results. Section 5 introduces our proposed mutation-based confidence smoothing solution and Section 6 presents its related evaluation. Section 7 discusses the limitation of this work and Section 8 concludes this work.

2 RELATED WORK

We review the related works from the perspectives of fault detection for deep neural networks and large language model testing.

2.1 Fault Detection for Deep Neural Networks

To save the labeling budget, multiple test selection methods [3, 35, 39, 41] have been proposed. A recent survey [14] reviewed test selection methods and divided them into fault detection methods, sampling-based model retraining methods, model selection methods, and performance estimation methods. We focus on the fault detection methods.

The early method DeepGini [9] has been proposed by Feng *et al.* to reveal inputs that are more likely been mispredicted by the model. DeepGini defined a new Gini score to measure the uncertainty of deep learning models on the inputs. Later on, inspired by the mutation testing in the conventional SE field, Chen *et al.* proposed to mutate both input samples and deep learning models and identify faults by computing the killing score. Their evaluation on more than 20 tasks demonstrated that mutation testing is useful for helping find *bugs* in deep learning models. Furthermore, Gao *et al.* proposed an adaptive test selection (ATS) method. ATS not only considers detecting faults but also tries to find diverse faults (diverse here means faults are from different categories) in deep learning models. More recently, Bao *et al.* first empirically proved that simple methods (i.e., output probability-based methods) perform well on fault detection. Then, they proposed a new method to compute the uncertainty scores by averaging the basic score of the input and scores from the neighbors of this input. Besides, Ma *et al.* conducted a comprehensive empirical study to explore the potential of test selection methods and active learning methods on fault detection. They found that MaxP is the best in terms of the correlation between the MaxP score and the misclassification. Besides the SE community, researchers from the ML community also contributed to this field. Dan *et al.* built the first benchmark for detecting misclassified data and found that just using the confidence score is already a strong baseline for this task. Li *et al.* proposed TestRank which uses graph neural networks to learn the difference between benign inputs and faults for fault detection.

Different from these works, our work is the first to explore the effectiveness of fault detection methods for large language models. We found that existing methods cannot well detect faults in LLMs and proposed the use of prompt mutation to further enhance these methods.

2.2 Large Language Model Testing

Large language models have been the most used deep learning models for most of the tasks. Comprehensively testing or evaluating the performance of LLMs becomes one of the most important directions in all the communities.

Chang *et al.* surveyed works that focus on evaluating LLMs [4]. They classified the evaluation objectives into seven categories, natural language processing, robustness/ethics/biases/trustworthiness, social science, natural science & engineering, medical applications, agent applications, and other applications. Evaluating LLMs for code which we are more interested in lies in the category of natural science & engineering. Xu *et al.* [34] conducted a comprehensive study to evaluate four series of LLMs on the code generation task using the HumanEval benchmark. They found that the performance of LLMs is affected by the parameter size of models and the training time. Liu *et al.* [21] propose a new code synthesis evaluation framework EvalPlus to measure the correctness of code generated by LLMs. Based on this framework, they found that the existing benchmark HumanEval is not rigorous enough. The performance (passk) of LLMs drops a lot when using the new benchmark. Besides, Ma *et al.* [22] evaluated the capability of LLMs to understand code syntax and semantics. The experimental results show that LLMs can understand the syntax structure of code, and perform code static analysis, but cannot approximate code dynamic behavior of code.

More recently, Yuan *et al.* [36] compared LLMs with conventional code models that are fine-tuned for specific code tasks. They found that in the zero-shot setting, instruction-tuned LLMs have competitive performance to fine-tuned code models. In the one-shot setting, the guidance by one-shot example sometimes harms the performance of LLMs. Du *et al.* [8] argued that existing works that evaluate the effectiveness of LLMs for code mainly focus on simple code tasks (function-level code synthesis) and built the first class-level code regeneration benchmark for the LLM evaluation. Based on their benchmark, they found that existing LLMs have significantly lower performance on class-level code generation tasks compared to method-level code generation tasks. There is still a big room to improve the ability of LLMs for code generation.

Unlike existing works that focus on evaluating the performance of LLMs from the perspective of preparing test cases, our work measures the capability of LLMs from another view and studies the problem of quickly identifying potential faults in LLMs to save the testing cost.

3 EMPIRICAL STUDY

In this work, we only focus on the problem of fault detection of LLMs. We consider both learning-based fault detection methods and output-probability-based methods in the literature. Based on the previous definition [14], given a LLM M , an unlabeled test set X_{test} , and a labeling budget $budget$, fault detection is to select a subset $\tilde{X}$ of X_{test} such that $\tilde{X}_{budget} = \arg \min_{\{X_i \subseteq X_{test} \wedge |X_i| = budget\}} \rho(M, X_i, Y_i)$ where Y_i are the labels corresponding to X_i , and $\rho(\cdot)$ is the performance measurement function. We follow the guidance of previous works [14, 15, 23] to select nine fault detection methods in our study. We exclude neuron coverage-based methods and surprise adequacy methods in our benchmark since the parameter size of LLMs is too big (>7B) and the coverage and adequacy score are difficult to compute.

3.1 Fault Detection Methods

Let $\mathcal{X}$ be a set of test inputs and $\mathcal{Y}$ be the corresponding label set, $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ denote a given test sample and its true label, respectively. Let $p_{y_i}(x)$ be the likelihood of x belonging to class $y_i \in \mathcal{Y}$ produced by the model $\mathcal{M}$.

- **Random Selection** is the basic selection criteria that test the model using randomly selected a subset of test data.
- **Max Probability (MaxP)** prioritizes data based on the maximum output probability $\text{Max}(p_y \mid y \in \mathcal{Y})$. The data sample with a lower MaxP score is more likely to be a fault.
- **DeepGini** defines a Gini score calculated by $\text{Gini}(x) = 1 - \sum_{y_i \in \mathcal{Y}} p_{y_i}^2(x)$ to measure the uncertainty of data. The data with higher Gini scores are treated as faults.
- **Entropy** selects data with the minimum Shannon entropy calculated using output probabilities.
- **Margin** measures the uncertainty of data based on the difference between the top-1 and top-2 output probabilities. A smaller difference indicates that the model is more difficult to distinguish the data between the two classes and the data should be considered faults.
- **Multiple-Boundary Clustering and Prioritization (MCP)** contains two steps. First, it divides the data space into different areas based on the decision boundary. Here, the decision boundary is approximated by the top-2 prediction probabilities. For example, give an input x and its top-2 output probabilities are p_{y_1} and p_{y_3} . Then this input is near the decision boundary (1, 3) and on the side of 1. MCP then selects data from different data spaces based on the score $\frac{p_{y_f}}{p_{y_s}}$, where y_f and y_s are the top-1 and top-2 probabilities.
- **Nearest Neighbor Smoothing (NNS)** first finds the neighbors and then smooths the output probabilities of each input using the outputs of neighbors. Finally, uncertainty-based fault detection methods such as DeepGini are employed to select the faults. In NNS, neighbors are searched by computing the distance between each extracted representation of the input. NNS requires the intermediate outputs of inputs to conduct data selection and, thus, cannot be used for closed-source LLMs such as GPT3.5.
- **Adaptive Test Selection (ATS)** first projects the top-3 maximum output probabilities to the space plane. After that, it computes the coverage of each input on the plane. Then, the coverage score is used to identify if the input is a fault or not based on its difference from the coverage of the whole test set. Note that ATS can only be used for classification tasks that have greater than three categories.
- **TestRank** initially extracts two features from the input data: 1) the output from the logits layer as intrinsic attributes, and 2) the graph information, which includes the cosine distance to other data points and the label of the data. Subsequently, a graph neural network (GNN) model is employed to assimilate the graph information and forecast the contextual attributes of the data. Finally, the contextual attributes and intrinsic attributes are amalgamated and inputted into a binary classification model to acquire proficiency in identifying failures. Comparable to the NNS method, TestRank also necessitates intermediate outputs for data selection and is not applicable to closed-source LLMs.

Except for the random selection and TestRank, all the other methods are purely based on the output probability. For the TestRank, the output probability is also a part of the information used for the detection. Thus, the prediction confidence of LLMs is important for fault detection.

3.2 Research Question

Our study plans to explore the effectiveness of existing fault detection methods on LLMs. Concretely, we answer the following two research questions:

Table 1. Datasets and models used in our study.

Task	Class Number	Test Size	Model	Accuracy
Clone Detection (CD)	2	200	CodeLLaMA	49.0%
			GPT3.5	75.5%
			GPT4	77.0%
Problem Classification (PC)	5	200	CodeLLaMA	17%
			GPT3.5	36%
			GPT4	100.0%
Sentiment Analysis (Sentiment)	3	150	LLaMA	89.3%
			GPT3.5	82.0%
			GPT4	90.0%
TagMyNews (TMN)	7	200	LLaMA	21.5%
			GPT3.5	60.0%
			GPT4	82.0%

- **RQ1:** *How confident are LLMs in their prediction?* Most (seven out of nine) of the fault detection methods are purely output uncertainty-based. Therefore, checking the prediction confidence produced by LLMs before running fault detection methods is necessary.
- **RQ2:** *How effective are fault detection methods for LLMs?* In this research question, we check if the existing methods demonstrated to be useful in classical deep learning models can perform well on LLMs.

3.3 Datasets and Models

In total, we consider four datasets including both natural language classification tasks and programming language classification tasks, and four types of LLMs covering both open-sourced models and closed-source models.

Clone Detection (CD): Utilizing the BigCloneBench [29] dataset, we randomly selected 200 items outside of 2 classes to investigate code clone detection. Models are evaluated on their ability to predict the presence of code clones (a probability score from 0 to 1, where 0 indicates no clone and 1 indicates a clone) between code snippet pairs, a critical task for enhancing software maintenance and development practices.

Problem Classification (PC): Based on the Java250 [27] dataset from the CodeNet Project, problem classification focuses on the classification of programming challenges on LLMs. A set of 200 samples across 5 distinct problem types was chosen in a way of a one-shot learning prompt. This task tests the model’s proficiency in categorizing coding problems using a singular example for each category, requiring output as a probability distribution across classes that sum to 1.

Sentiment Analysis (Sentiment): For this task, 150 samples were extracted from the Sentiment Analysis of IMDB Movie Reviews dataset ². The challenge for models is to predict the sentiment of movie reviews, negative, neutral, and positive, ranging from 0 to 1 without any prior examples (zero-shot learning), evaluating the capacity of models for natural language understanding and emotional tone assessment.

TagMyNews (TMN): Using the TagMyNews Dataset, we randomly picked 200 articles with seven categories, which are sourced from RSS feeds of popular newspapers [32]. The task aims to assess the efficacy of the model in accurately categorizing news articles based on their textual content utilizing a one-shot learning framework. The expected output schema is similar to the problem classification task.

LLaMA and CodeLLaMA. LLaMA [31] is known for its auto-regressive architecture aimed at chat applications. The parameters of the series of LLaMA models range from 7 billion to 70 billion

²Autolabel, <https://github.com/refuel-ai/autolabel>

and are learned from publicly available online data amounting to 2 trillion tokens. Besides, LLaMA models have been fine-tuned with a combination of supervised fine-tuning (SFT) and reinforcement learning with human feedback (RLHF), focusing on aligning the models to human preferences for helpfulness and safety. CodeLLaMA [28] is one variant of LLaMA that specifically designed for programming contexts. This adaptation involves further fine-tuning LLaMA on datasets specific to coding, thereby boosting their code-related performance capabilities. Our experiments are carried out using the *meta-llama/Llama-2-7b-chat-hf* and *codellama/CodeLlama-7b-Instruct-hf* models, available on the public Hugging Face platform.

GPT3.5 and GPT4 [1]. GPT3.5 is a fine-tuned version of GPT3 developed in January 2022. It is built on Transformer architecture while it emphasizes decoder-only mechanisms. GPT4, which is known as the state-of-the-art LLM, has a substantial leap in capability and complexity compared to GPT-3.5. Both GPT3.5 and GPT4 are closed-source LLMs, we employ *gpt-3.5-turbo-0125* and *gpt-4-0125-preview* models in our experiments through API access, specifically within text chat application contexts.

Table 1 presents the details of our used datasets and models. We found that in some cases (marked by red), the performance of LLMs is too bad (close to random guess) and too good (with perfect accuracy). For these cases, we only evaluate their prediction confidence (RQ1) and do not conduct fault detection (RQ2) on them.

3.4 Evaluation Metrics.

In RQ1, we follow the previous work [12] to evaluate the confidence of LLMs in their prediction using metrics *Prediction Confidence (Confidence)* and *Expected Calibration Error (ECE)*. Roughly speaking, *Confidence* is computed by probabilities associated with the predicted label. *ECE* first splits the range of the prediction score into M equally-spaced intervals (denoted as I_m , $m \in \{1, \dots, M\}$) and then computes the weighted average of the bins' accuracy/confidence. A lower ECE score indicates that the model is better calibrated, i.e., the prediction probabilities can better represent the true correctness of models.

Definition 1 (Prediction Confidence (Confidence)).

$$Confidence(x) = \max(\{p_{y_i}(x) | 0 < i \leq N\})$$

where N is the number of classes.

Definition 2 (Expected Calibration Error (ECE)). Given a test set X , let $B_m \subseteq X$ be the inputs whose prediction confidence falls into the m^{th} interval $I_m = (\frac{m-1}{M}, \frac{m}{M}]$, $Acc(B_m)$ be the accuracy of the inputs B_m , and $Avg_Conf(B_m) = \frac{1}{|B_m|} \sum_{x \in B_m} Confidence(x)$ be the average confidence within B_m , then the expected calibration error on X is defined as:

$$ECE(X) = \sum_{m=1}^M \frac{|B_m|}{|X|} |Acc(B_m) - Avg_Conf(B_m)|$$

In RQ2, we follow the previous work [19] and evaluate the effectiveness of fault detection methods using the metric of Test Relative Coverage (TRC) which is calculated by the percentage of the number of faults divided by the labeling budget or the total number of faults whichever is minimal.

Definition 3 (Test Relative Coverage (TRC)). Given a test set X and a labeling budget *budget*, the test relative coverage on X is defined as:

$$TRC(X) = \frac{|\tilde{X}_{faults}|}{Min(budget, |X_{faults}|)}$$

where $\tilde{X}_{faults}$ represents the faults within the selected test set $\tilde{X}_{budget}$ and X_{faults} represents the faults in the whole test set X . A higher TRC value indicates the better performance of the fault detection method.

3.5 Configurations

Configuration of fault detection methods. NNS and TestRank need to compute the distance between data samples and require the representation of the samples. In this work, we directly use the input embeddings extracted from LLMs as the representation. Besides, TestRank is a learning-based method where training data is required for fault detection. For the clone detection, problem classification, and TagMyNews tasks, we randomly select 200 data from the original datasets as the training data. For the sentiment analysis dataset, as the original dataset only contains 200 data samples and 150 of them are used as test data, we collect the remaining 50 samples as the training data. For the data labeling process, we set the labeling budgets as 10%, 30%, and 50%. *Configuration of LLMs.* In our experiments, we set $top_p=0.9$ for LLaMA/CodeLLaMA and $top_p=1$ for GPT3.5/GPT4 which are the default settings used by LLMs. top_p controls the accumulation of token probabilities to select the next word for all LLMs. We set the window size of the token context as $max_length=10k$ for LLaMA/CodeLLaMA to match our test data's token length. The window sizes of GPT3.5 and GPT4 are set as 16385 and 128000 respectively, using the default settings.

3.6 Implementation and Environment

For the fault detection methods, we reuse the official implementations provided by the original papers and modify them to fit our tasks. For the LLaMA and CodeLLaMA models, we use the resources provided by Hugging Face³ in this work. For GPT3.5 and GPT4 models, we use the OpenAI's API to access the models and collect the results. In total, it costs 267 US dollars to run GPT-related experiments. We conduct all LLaMA-related experiments on a 2.6 GHz Intel Xeon Gold 6132 CPU with an NVIDIA Tesla V100 16G SXM2 GPU.

4 EMPIRICAL RESULTS

4.1 RQ1: Prediction Confidence

First, we check whether LLMs can confidently predict the data in our studied tasks. Figure 1 depicts the distribution (number of data in each confidence interval) of the prediction confidence produced by LLMs, where we set the number of intervals M as 30. The first conclusion we can draw is that the series of GPT models are more confident with their predictions than the series of LLaMA models. On average, the confidence of LLaMA, CodeLLaMA, GPT3.5, and GPT4 on these four datasets are 0.5338, 0.6993, 0.7701, and 0.8727, respectively. Comparing the accuracy of LLMs and their average confidence (which is a notion of miscalibration [12]), we found that existing LLMs are not well-calibrated. For example, for CodeLLaMA on the clone detection task, the accuracy and average confidence are 49% and 0.758, respectively. This means CodeLLaMA is overconfident with its prediction even though it has poor performance. Overall, except for sentiment analysis, LLMs are overconfident in the other three tasks with 21.36% higher confidence than the true accuracy. Besides, ECE demonstrated that GPT4 is better calibrated than other LLMs in all our considered tasks. The Reliability diagrams [12] that visualize the model calibration can be found in Appendix A.

³Hugging Face, <https://huggingface.co/>

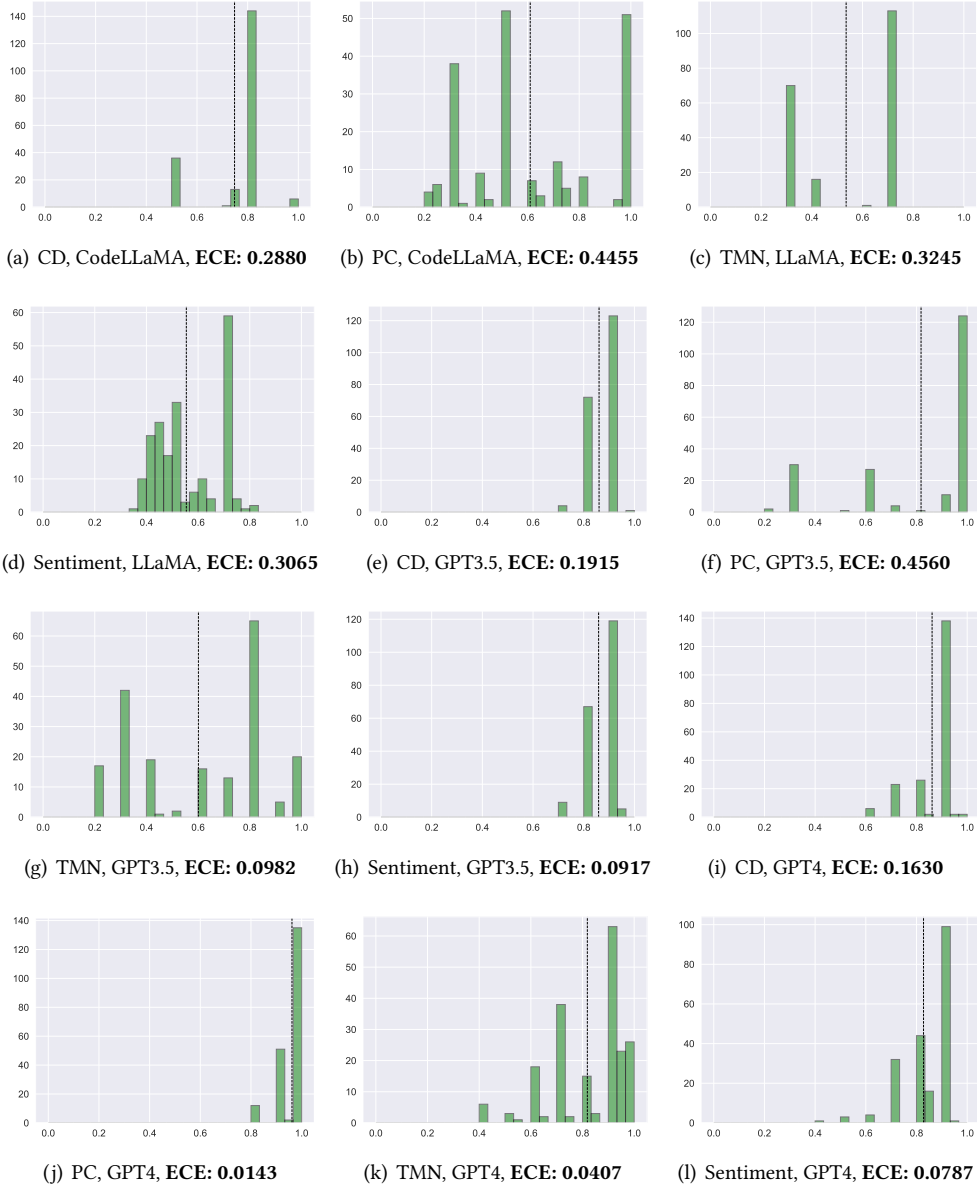


Fig. 1. Distribution of prediction confidence and ECE for LLMs. The dashed line indicates the average confidence.

Additionally, the results show that prediction confidence is concentrated in certain ranges for many cases. For example, for the clone detection - GPT3.5, confidence only falls into four ranges. This means the model has similar confidence to many different inputs, in turn, the output-based fault detection methods (e.g., MaxP) could produce similar uncertainty scores to these inputs.

Table 2. Test relative coverage of each method on LLaMA. The best method is highlighted in blue.

	Budget	Random	Gini	Entropy	MCP	MaxP	Margin	ATS	NNS	TestRank
Sentiment Analysis	10%	0.0667	0.0667	0.0667	0.1333	0.0000	0.1333	0.0667	0.0000	0.0667
	30%	0.4375	0.0625	0.1250	0.5000	0.0000	0.1250	0.1250	0.1250	0.0625
	50%	0.4375	0.3125	0.2500	0.5625	0.2857	0.4375	0.3750	0.1875	0.2500
	Average	0.3139	0.1472	0.1472	0.3986	0.0952	0.2319	0.1889	0.1042	0.1264

Considering the other two cases that have poor performance reported in Table 1, CodeLLaMA for problem classification and LLaMA for TagMyNews. We found they have confused confidence on these tasks as shown in Figure 5(b) and Figure 5(c). For instance, in Figure 5(b), the most frequent confidence score is 0.5, and in Figure 5(c), there are no confidence scores that are higher than 0.8. This phenomenon indicates that CodeLLaMA and LLaMA cannot handle problem classification and news classification tasks at all.

Finally, we found that GPT4 has perfect accuracy (100%) on the problem classification task with high average confidence (0.9827 on average) and a low ECE score of 0.0143. We conjecture there is a data leakage problem of our test data, i.e., our used problem classification dataset is a part of the training data of GPT4. As there are no faults in GPT4 for the problem classification task, we exclude this model from the following fault detection study.

Answer to RQ1: LLMs are not well-calibrated and overconfident in clone detection, problem classification, and news classification tasks. Besides, prediction confidence is concentrated in a few intervals, indicating LLMs have similar confidence to most of the inputs.

4.2 RQ2: Effectiveness of Fault Detection

In this RQ, we explore the performance of existing fault detection methods on LLMs. As mentioned in Section 3.3, we exclude models that have poor performance (nearly random guesses) and perfect performance (100% accuracy) on the studied tasks.

Table 2 and Table 5 summarize the results. The first conclusion we can draw from the results is that the existing fault detection methods cannot significantly find faults in LLMs compared to their performance on classical deep learning models [19] where the best method can achieve a 0.9532 TRC score. For example, considering the detection performance on the LLaMA model, only the best method has a higher TRC score (0.3986) than random selection (0.3139). This means all the other well-designed methods perform worse than random guesses. Considering the results on GPT3.5 and GPT4 models, this phenomenon has been alleviated. All the methods perform better than random selection. However, the best methods on GPT3.5 and GPT4 have only 0.5771 and 0.6489 TRC scores. There is still a big gap between the existing performance and the idea performance, TRC=1.

Then, comparing each method, we found that no method can consistently perform better than the others. The performance of fault detection methods on LLMs highly depends on the datasets and models. However, it is interesting that when the performance of LLMs becomes better, the fault detection methods become more effective. For example, the TRC scores produced by each method on GPT4 (on average 0.6114) are much higher than the TRC scores on LLaMA (on average 0.3074). The reason is that existing methods that are based on output probabilities highly rely on the confidence of models in their outputs. As GPT4 is more confident than LLaMA, the fault detection methods can perform better on GPT4 accordingly.

Table 3. Test relative coverage of each method on GPT. The best method is highlighted in blue.

		Budget	Random	Gini	Entropy	MCP	MaxP	Margin	ATS
GPT3.5	Clone Detection	10%	0.1500	0.3000	0.4500	0.3500	0.3000	0.3000	-
		30%	0.3265	0.4694	0.4490	0.4082	0.4694	0.4694	-
		50%	0.5102	0.6327	0.6122	0.6122	0.6327	0.6327	-
	Problem Classification	10%	0.7000	0.8000	0.7500	0.7000	0.8000	0.8500	0.6500
		30%	0.7167	0.7667	0.6500	0.6333	0.7833	0.7667	0.6833
		50%	0.5900	0.7000	0.6100	0.6400	0.7000	0.7000	0.6700
	TagMyNew	10%	0.7000	0.5500	0.5500	0.6000	0.5500	0.5500	0.4000
		30%	0.4000	0.5333	0.5333	0.4500	0.5500	0.5500	0.4889
		50%	0.4375	0.5625	0.5500	0.5000	0.6000	0.6625	0.8444
	Sentiment Analysis	10%	0.0667	0.2667	0.1333	0.2667	0.2667	0.3333	0.0000
		30%	0.3704	0.3704	0.4074	0.4444	0.4444	0.4074	0.2222
		50%	0.5926	0.6667	0.7037	0.5556	0.6667	0.7037	0.5185
	Average		0.4634	0.5515	0.5333	0.5134	0.5636	0.5771	0.4975
GPT4	Clone Detection	10%	0.4000	0.5000	0.5000	0.2500	0.5000	0.2000	-
		30%	0.3478	0.5652	0.5435	0.5217	0.5652	0.3478	-
		50%	0.5435	0.6522	0.6522	0.6739	0.6522	0.5652	-
	TagMyNews	10%	0.3000	0.4500	0.3500	0.4000	0.4500	0.4500	0.4000
		30%	0.3333	0.6667	0.6667	0.5000	0.6389	0.6667	0.7222
		50%	0.5278	0.8611	0.8611	0.7778	0.8611	0.8611	0.8333
	Sentiment Analysis	10%	0.0667	0.3333	0.5333	0.3333	0.3333	0.3333	0.2667
		30%	0.4667	0.6667	0.7333	0.7333	0.6667	0.6667	0.3333
		50%	0.6000	0.9333	1.0000	0.9333	0.9333	0.9333	0.5333
	Average		0.3984	0.6254	0.6489	0.5693	0.6223	0.5582	0.5148

Answer to RQ2: Existing fault detection methods are not effective in detecting faults in LLMs and there are no methods that can consistently perform better than the others. Methods have better performance on LLMs that have higher accuracy (GPT4) than those that have lower accuracy (GPT3.5 and LLaMA).

5 MUCS

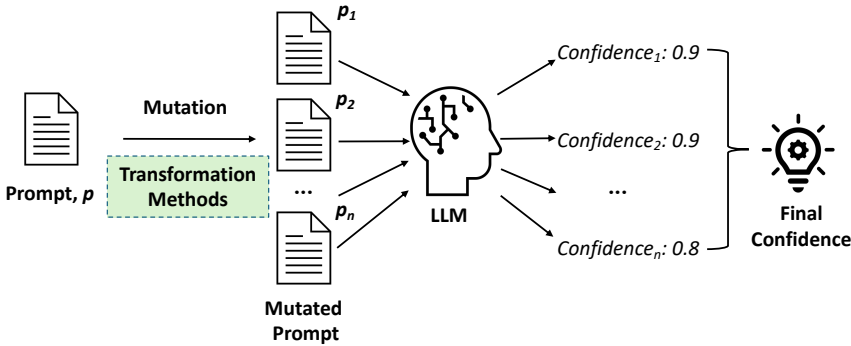


Fig. 2. Workflow of MuCS.

Since the concentrated confidence will harm the performance of output-based fault detection methods (those methods will give similar uncertainty scores to the inputs), we need to diversify

Algorithm 1: MuCS

```

Input   :  $p$ : input prompt
            $OPs$ : mutation operators
            $M$ : LLM
            $K$ : perturbation size
            $n$ : number of generated mutants

Output :  $C_{smooth}$ : smoothed confidence

1  $Cs = []$ 
2 for  $i = 0 \rightarrow n$  do
    /* mutants generation */
3   for  $j = 0 \rightarrow K$  do
4      $OP = \text{RandomSelection}(OPs)$ 
5      $p' = OP(p)$ 
6      $p = p'$ 
7   end
    /* confidence calculation using Definition 1 */
8    $Cs.append(\text{Confidence}(M(p)))$ 
9 end
    /* confidence averaging */
10  $C_{smooth} = \text{Mean}(Cs)$ 
11 return  $Cs$ ;

```

the prediction confidence of LLMs to enhance fault detection. It is challenging since we cannot get the internal information (e.g., output logits) of closed-source LLMs to smooth the prediction confidence. Inspired by previous works [20, 33] that showed the potential of testing deep learning models by mutation analysis (one uses mutation killing score to detect faults and the other one uses mutation analysis to improve the performance of code models), we propose MuCS, a simple yet effective black-box solution to enhance fault detection by smoothing the prediction confidence from the input-level using prompt mutation.

Figure 2 presents the overall workflow of our proposed solution. It has two main steps, 1) prompt mutation, and 2) confidence smooth. Specifically, given an LLM and an input prompt p , we first utilize transformation methods to mutate the inputs and generate n mutated prompts, $p_1, p_2, \dots, p_n$. For NLP tasks, we use text augmentation methods [24] (e.g., token random deletion) to transform prompts into new ones. For the code tasks, we consider both text augmentation methods and code refactoring methods [2] (e.g., add new *Print()* functions) for the prompt mutation. After that, we feed all the mutants to the LLM and collect the output confidence $Confidence_1, Confidence_2, \dots, Confidence_n$ produced by the LLM. Finally, a confidence smooth method is employed to compute the *final confidence* of the LLM on the input. In this work, we use the straightforward method, average, to smooth the confidence. Algorithm 1 summarizes our mutation-based confidence smoothing solution.

5.1 Mutation Operators

Two types of mutation operators have been used, text augmentation methods and code refactoring methods. In total, five text augmentation methods (including Synonym Replacement, Random Deletion, Random Insertion, Random Swap, and Punctuation Insertion), and four code refactoring methods (including Print Adding, Local Variable Adding, Dead If Adding, and Duplication) have

been used in MuCS. Given a prompt p with the length of works k , each mutation operator is defined as follows.

- **Synonym Replacement (SP)** randomly selects n words in p and replaces them as their synonyms, where $n < k$. Here, the synonyms are searched from the WordNet [25]. We use the default setting of TextAugment [24], $n = 1$ in MuCS.
- **Random Deletion (RD)** randomly removes words in the p with a probability. Specifically, RD assigns a random probability pro uniformed from 0 to 1 to each word in p . Given a probability threshold T , if pro is less than T , the corresponding word will be removed. We use the default setting of MixCode [7], $T = 0.01$ in MuCS.
- **Random Insertion (RI)** randomly selects n words in p first where $n < k$. Then, it finds the synonyms of selected words and inserts them into the random locations of p . We use the default setting of MixCode, $n = 1$ in MuCS.
- **Random Swap (RW)** randomly swaps two words in p .
- **Punctuation Insertion (PI)** randomly selects n punctuation (e.g., !) and inserts them into the random locations of p , where $n < k$. We use the default setting of MixCode, $n = 1$ in MuCS.
- **Print Adding (PA)** randomly selects a line in the code and adds a print function with randomly generated content below it.
- **Local Variable Adding (LVA)** randomly selects a line in the code and defines an unused local variable below it.
- **Dead If Adding (DIA)** randomly selects a line in the code and then adds an unreachable if statement below it.
- **Duplication** randomly copies an assignment in the code and inserts it next to this assignment.

During the mutant generation process, we randomly select K mutation operators to mutate p and generate its mutant p' .

6 EVALUATION

6.1 Research Question

To check whether MuCS can help fault detection or not, we first analyze the effectiveness of existing fault detection methods using MuCS. Then, we explore the reason why MuCS enhances fault detection by analyzing its influences on the prediction confidence of LLMs.

- **RQ3:** *How effective is MuCS in enhancing fault detection?* In this research question, we compare the effectiveness of fault detection methods with and without using MuCS to analyze the usefulness of our proposed solution.
- **RQ4:** *How does MuCS affect the outputs of the LLMs?* We want to explore how MuCS influences the outputs of LLMs to check why it works on fault detection.

6.2 Evaluation Setup

We set the mutation times as 10 for all tasks. That is, we generate 10 mutants for each input prompt. We set K (the number of mutation operators used) as 3 for all the tasks except for the clone detection. The reason is that we found the performance of LLMs on the clone detection task drops significantly (nearly random guesses, e.g., all the predicted labels of GPT3.5 are 0) when $K = 3$ and $K = 2$. Therefore, we set $K = 1$ for the clone detection task. For other tasks, LLMs have similar accuracy on the mutated datasets to the original ones (with less than 5% difference). For the other settings, we use the same settings as Section 3.

Table 4. Test relative coverage (relative improvement compared to Table 2) of each method on LLaMA after prompt mutation-based confidence smooth. The best method is highlighted in blue. Improvement indicates the average relative improvement compared to Table 2. * denotes the original TRC score in Table 2 is 0 and it is skipped in relative comparison.

Budget	Gini-M	Entropy-M	MCP-M	MaxP-M	Margin-M	ATS-M	NNS-M	TestRank-M	BALD
10%	0.0000(↓ 100.00%)	0.0667(− 0.00%)	0.1333(− 0.00%)	0.0000(− *)	0.2667(↑ 100.08%)	0.0667(− 0.00%)	0.0000(− *)	0.0000(↓ 100.00%)	0.2667
30%	0.1250(↑ 100.00%)	0.1250(− 0.00%)	0.5000(− 0.00%)	0.2500(− *)	0.3125(↑ 150.00%)	0.2500(↑ 100.00%)	0.0625(↓ 50.00%)	0.1250(↑ 100.00%)	0.5625
50%	0.5625(↑ 80.00%)	0.5625(↑ 125.00%)	0.7500(↑ 33.33%)	0.5625(↑ 96.88%)	0.6250(↑ 42.86%)	0.3750(− 0.00%)	0.3750(↑ 100.00%)	0.5625(↑ 125.00%)	0.8125
Average	0.2292(↑ 26.67%)	0.2514(↑ 41.67%)	0.4611(↑ 11.11%)	0.2708(↑ 96.88%)	0.4014(↑ 97.64%)	0.2306(↑ 33.33%)	0.1458(↑ 25.00%)	0.2292(↑ 41.67%)	0.5472

Table 5. Test relative coverage (relative improvement compared to Table 3) of each method on GPT after prompt mutation-based confidence smooth. Improvement indicates the average relative improvement compared to Table 3.

		Budget	Gini-M	Entropy-M	MCP-M	MaxP-M	Margin-M	ATS-M	BALD
GPT3.5	Clone Detection	10%	0.4500(↑ 50.00%)	0.4500(− 0.00%)	0.4500(↑ 28.57%)	0.4500(↑ 50.00%)	0.4500(↑ 50.00%)	-	0.4500
		30%	0.5306(↑ 13.04%)	0.5102(↑ 13.63%)	0.5306(↑ 29.99%)	0.5306(↑ 13.04%)	0.5306(↑ 13.04%)	-	0.3265
		50%	0.6939(↑ 9.67%)	0.5510(↓ 10.00%)	0.5918(↓ 3.33%)	0.6939(↑ 9.67%)	0.6939(↑ 9.67%)	-	0.5714
	Problem Classification	10%	0.8000(− 0.00%)	0.8000(↑ 6.67%)	0.5500(↓ 21.43%)	0.8500(↑ 6.25%)	0.8000(↑ 5.88%)	0.8500(↑ 30.77%)	0.8500
		30%	0.8333(↑ 8.69%)	0.7833(↓ 20.51%)	0.6500(↑ 2.64%)	0.8500(↑ 8.52%)	0.8500(↑ 10.86%)	0.7667(↑ 12.21%)	0.8500
		50%	0.8000(↑ 14.29%)	0.8300(↑ 36.07%)	0.6900(↑ 7.81%)	0.8000(↑ 14.29%)	0.7700(↑ 10.00%)	0.7200(↑ 7.46%)	0.7700
	TagMyNew	10%	0.6000(↑ 9.09%)	0.6500(↑ 18.18%)	0.4000(↓ 33.33%)	0.7000(↑ 27.27%)	0.6000(↑ 9.09%)	0.7000(↑ 75.00%)	0.7500
		30%	0.6000(↑ 12.51%)	0.6000(↑ 12.51%)	0.4500(− 0.00%)	0.6167(↑ 12.13%)	0.5833(↑ 6.05%)	0.6042(↑ 23.58%)	0.6167
		50%	0.6875(↑ 22.22%)	0.6625(↑ 20.45%)	0.5750(↑ 15.00%)	0.7000(↑ 16.67%)	0.6750(↑ 1.89%)	0.7917(↑ 6.24%)	0.6500
	Sentiment Analysis	10%	0.3333(↑ 24.97%)	0.4000(↑ 200.08%)	0.5333(↑ 99.96%)	0.2667(− 0.00%)	0.3333(− 0.00%)	0.4000(− *)	0.3333
		30%	0.5926(↑ 59.99%)	0.5926(↑ 45.46%)	0.7037(↑ 58.35%)	0.5556(↑ 25.02%)	0.6667(↑ 63.65%)	0.5926(↑ 166.70%)	0.5556
		50%	0.9259(↑ 38.88%)	0.8889(↑ 26.32%)	0.8148(↑ 46.65%)	0.9259(↑ 38.88%)	0.9259(↑ 31.58%)	0.6667(↑ 28.58%)	0.6296
Average	0.6539(↑ 21.95%)	0.6432(↑ 32.49%)	0.5783(↑ 19.24%)	0.6616(↑ 18.48%)	0.6566(↑ 16.66%)	0.6769(↑ 42.26%)	0.6128		
GPT4	Clone Detection	10%	0.4500(↓ 10.00%)	0.4500(↓ 10.00%)	0.4000(↓ 60.00%)	0.4500(↓ 10.00%)	0.3000(↓ 50.00%)	-	0.2500
		30%	0.6304(↑ 11.54%)	0.6304(↑ 15.99%)	0.6304(↑ 20.84%)	0.6304(↑ 11.54%)	0.2391(↓ 31.25%)	-	0.2222
		50%	0.7826(↑ 19.99%)	0.7391(↑ 13.32%)	0.7826(↑ 16.13%)	0.7826(↑ 19.99%)	0.4565(↓ 19.23%)	-	0.4222
	TagMyNews	10%	0.4500(− 0.00%)	0.4000(↑ 14.29%)	0.4000(− 0.00%)	0.4000(↓ 11.11%)	0.4000(↓ 11.11%)	0.4000(− 0.00%)	0.5000
		30%	0.5556(↓ 16.66%)	0.5278(↓ 20.83%)	0.6389(↑ 27.78%)	0.6389(− 0.00%)	0.6389(↓ 4.17%)	0.7222(− 0.00%)	0.5833
		50%	0.8889(↑ 3.23%)	0.8056(↓ 6.45%)	0.7778(− 0.00%)	0.9167(↑ 6.46%)	0.9167(↑ 6.46%)	0.8611(↑ 3.34%)	0.7222
	Sentiment Analysis	10%	0.4000(↑ 20.01%)	0.4000(↓ 25.00%)	0.4000(↑ 20.01%)	0.4000(↑ 20.01%)	0.4000(↑ 20.01%)	0.4667(↑ 74.99%)	0.3333
		30%	0.8667(↑ 30.00%)	0.8667(↑ 18.19%)	0.8000(↑ 9.10%)	0.8667(↑ 30.00%)	0.8667(↑ 30.00%)	0.6667(↑ 100.03%)	0.5333
		50%	0.9333(− 0.00%)	0.9333(↓ 6.67%)	0.9333(− 0.00%)	0.9333(− 0.00%)	0.9333(− 0.00%)	0.7333(↑ 37.50%)	0.5333
	Average	0.6619(↑ 6.46%)	0.6392(↓ 0.79%)	0.6403(↑ 17.09%)	0.6687(↑ 7.43%)	0.5724(↑ 4.52%)	0.6417(↑ 35.98%)	0.4555	

6.3 RQ3: Effectiveness of MuCS

First, we explore whether the MuCS can enhance the existing fault detection methods or not. Before that, since the input mutation can produce multiple predictions from LLMs which is similar to the dropout uncertainty [10]. We introduce another fault detection method baseline which is based on dropout uncertainty, Bayesian Active Learning by Disagreement (BALD) [13].

Bayesian Active Learning by Disagreement (BALD) defines the uncertainty of an input by the disagreement of the outputs produced by LLMs on the mutants of this input. The disagreement score is calculated by $1 - \frac{\text{count}(\text{mode}(y^1, \dots, y^T))}{T}$, where y_i is the predicted label of mutant p_i and T is the number of mutants. A higher BALD score indicates the input is more likely to be a fault.

Table 4 and Table 5 summarize the TRC scores of various methods enhanced by MuCS (with -M as a suffix), and the relative difference between the scores with and without using MuCS. The results demonstrate that the MuCS significantly boosts the performance of fault detection. In terms of the average TRC scores of each method on different tasks, in 19 out of 20 cases, the performance of existing fault detection methods has been increased with TRC improvements ranging from 0.0366 to 0.1794. Considering the average relative improvement, confidence smooth can enhance the performance of existing methods by up to 97.64% (LLaMA for sentiment analysis, method: Margin). Therefore, we can conclude that MuCS is effective in boosting the existing fault detection methods on LLMs.

Comparing the improvements across different LLMs, we found that improvements in GPT4 are weaker than improvements in other LLMs. On average, the improvements in LLaMA, GPT3.5,

Table 6. Average confidence, ECE, and variance of prediction distribution before and after confidence smoothing.

Model	Confidence ($\downarrow$)		ECE ($\downarrow$)		Diversity ($\downarrow$)	
	Before	After	Before	After	Before	After
CD-GPT35	0.8535	0.8863	0.1915	0.2563	584.96	655.29
CD-GPT4	0.8575	0.8665	0.1630	0.1638	696.62	542.42
PC-GPT35	0.7659	0.7349	0.4560	0.3449	432.22	40.89
TMN-GPT35	0.6013	0.5981	0.0982	0.1030	205.36	24.69
TMN-GPT4	0.8187	0.7571	0.0407	0.0846	196.56	62.29
Sentiment-LLaMA	0.5476	0.5626	0.3065	0.3041	92.67	67.00
Sentiment-GPT35	0.8596	0.8153	0.0917	0.1180	335.87	104.87
Sentiment-GP4	0.8320	0.8059	0.0787	0.1093	230.60	93.33
Average change	$\uparrow$ 1.78%		$\downarrow$ 4.04%		$\uparrow$ 42.67%	

and GPT4 are 37.63%, 41.41%, and 11.78%, respectively. We conjecture that the reason for this phenomenon is that GPT4 is more powerful and more resilient to the input mutation. Thus, the smooth is less useful for GPT4. To check our assumption, we compute the average variance of the prediction output probability changes after prompt mutation of each LLM. The results show that the probability changes of GPT are negligible. For example, for the sentiment analysis task, the variance of the probability changes of LLaMA, GPT3.5, and GPT4 are 0.0047, 0.0158, and 0.0041, respectively.

Finally, even though the mutation-based confidence smooth can enhance the fault detection of LLMs, the best methods only have 0.5472, 0.6769, and 0.6687 TRC scores in LLaMA, GPT3.5, and GPT4, respectively. There is still a big room between the TRC scores achieved by existing methods and the ideal performance.

Answer to RQ3: MuCS significantly enhances the effectiveness of existing fault detection methods by up to 97.62% in terms of the TRC score.

6.4 RQ4: MuCS Affected Prediction Confidence

We further study how MuCS affects the prediction of LLMs and whether it diversifies the prediction confidence or not. Figure 5 depicts the confidence of LLMs on the datasets before and after confidence smoothing and Table 6 summarizes the average confidence and ECE scores. The results show no big difference between the average confidence of LLMs and ECE scores before and after smoothing. This finding is consistent with the conclusion from a previous work [42] that states the effectiveness of fault detection methods is not directly related to the calibration of the model (most confidence calibration methods are useless or harmful for fault detection).

Then, we check whether MuCS can increase the diversity of confidence distribution or not. We compute the variance of the number of confidence scores in each range to check this diversity. The results presented in Table 6 demonstrated that, except for the clone detection tasks, LLMs have more diverse predictions on the datasets with an average 42.67% diversity improvement. Taking Figure 6(b) as an example, the confidence scores of GPT3.5 are concentrated in 1.0 which is highly contradictory to its accuracy (36%). After the mutation-based smooth, confidence scores are spread throughout the distribution which is more useful to help understand the capability of LLMs on this task.

Answer to RQ4: MuCS significantly increases the diversity of confidence distribution of LLMs with an average 42.67% improvement and eliminates the concentrated confidence problem.

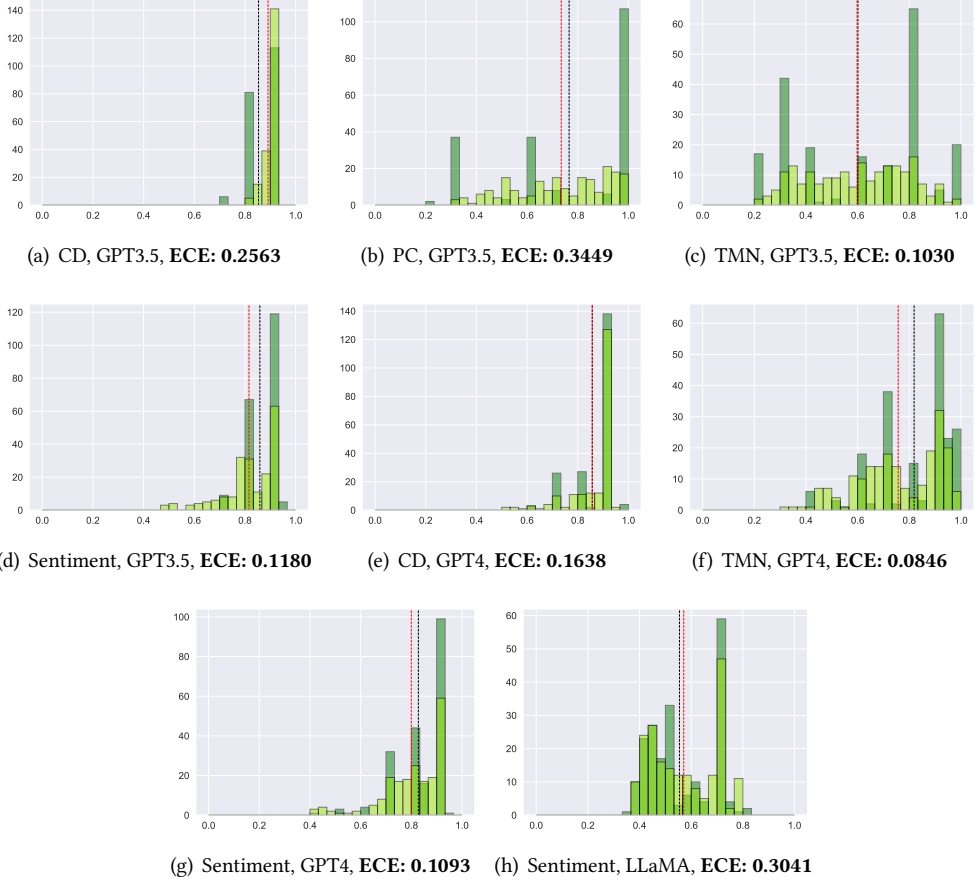


Fig. 3. Distribution of prediction confidence and ECE for LLMs. **Histogram**: confidence before smoothing. **Histogram**: confidence after smoothing. Black (Red) dashed line: average confidence before (after) smoothing.

7 DISCUSSION

7.1 Limitation and Future Direction

Limitation. The potential limitations are that 1) we only consider classification tasks. Non-classification tasks such as question answering are not included in this work. The reason is that most of the existing fault detection methods are designed for classification tasks only. Even though there also exist methods that can be employed for non-classification tasks, such as PRIMA [33], applying them for LLMs is not practical. PRIMA needs to mutate deep learning models, however, as LLMs are much larger than classical deep learning models, it is difficult to mutate LLMs and save the mutants. As a result, only classification tasks are studied in our work. 2) For a given input, our proposed mutation-based confidence smooth solution needs to access LLMs n times, thus, the money consumption is n times than the normal way when we test closed-source LLMs.

Future direction. As LLMs (such as ChatGPT) become the most used deep learning models nowadays, the most important task people would use is the generation tasks, e.g., chatting using ChatGPT. One of the most important directions in this field is to propose fault detection methods

for non-classification tasks, especially generation tasks, and then use these methods to help reveal faults in LLMs. Some metrics [16, 17] have been designed for measuring the uncertainty in LLMs, how to combine these uncertainty metrics with existing methods could be the first try.

7.2 Threats to Validity

The internal threat lies in the implementation of fault detection methods, LLMs, and prompt mutation methods. The implementation of each fault detection method comes from the official project provided original paper. The implementation of LLMs comes from the famous open-source project Hugging Face. The code of text mutation also comes from the commonly used project TextAugment [24]. The implementation of code refactoring methods modified from the previous work [7].

The external threat comes from our studied tasks, datasets, LLMs, and fault detection methods. As mentioned in Section 7.1, we only consider the classification tasks in this work. For the tasks, we consider tasks from both NLP and SE fields. Two traditional NLP tasks and two programming tasks are included in our work. For the studied LLMs, we consider both open-source LLMs, e.g., LLaMA, and closed-source LLMs (GPT4), where GPT4 is recognized as the SOTA LLM currently. Besides, another threat that comes from the datasets and models is that there might be data leakage issues in LLMs. For example, we found GPT4 has 100% accuracy on the problem classification task which means our test set is likely in the training set of GPT4. However, it is difficult to check this issue as GPT3.5 and GPT4 are closed-source LLMs. We can only exclude the cases from the analysis of the accuracy of LLMs. For the fault detection methods, we follow the previous works [14, 15, 23] and explore 10 fault detection methods including both learning-based methods and output-based methods in our study. These methods are from different communities, for example, DeepGini is from the SE community and TestRank is from the ML community. We believe our studied objectives are representative and diverse enough and the conclusion drawn from this work can be generalized to other similar cases.

The construct threat could be the hyperparameter settings and the non-determinism of LLMs. We directly use the default settings suggested by Hugging Face and OpenAI's API to build or access LLMs. Besides, we release the datasets and prompts used in our experiments to support to reproduce results reported in our work and for future research.

8 CONCLUSION

In this work, we first explored the potential of fault detection methods for LLMs. Based on the experiments on four tasks including NLP and code tasks, four LLMs, and nine fault detection methods, we found LLMs are overconfident with their predictions and current fault detection methods are not effective in revealing faults in LLMs. To enhance the detection performance, we proposed MuCS, which uses mutation analysis to smooth the prediction confidence of LLMs. Specifically, we utilized text transformation and code refactoring techniques to mutate the inputs and collect the averaged output probabilities of all mutants as the final prediction confidence of LLMs. Evaluation results demonstrated that MuCS significantly increased the diversity of output distribution of LLMs and enhanced the effectiveness of fault detection by up to 97.64% in terms of test relative coverage score.

REFERENCES

- [1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

- [2] Jihad Al Dallah and Anas Abidin. 2017. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering* 44, 1 (2017), 44–69.
- [3] Hamzah Al-Qadasi, Yliès Falcone, and Saddek Bensalem. 2023. DeepAbstraction++: Enhancing Test Prioritization Performance via Combined Parameterized Boxes. In *International Conference on Bridging the Gap between AI and Reality*. Springer, 77–93.
- [4] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2023. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology* (2023).
- [5] Nolan Dey, Gurpreet Gosal, Hemant Khachane, William Marshall, Ribhu Pathria, Marvin Tom, Joel Hestness, et al. 2023. Cerebras-gpt: Open compute-optimal language models trained on the cerebras wafer-scale cluster. *arXiv preprint arXiv:2304.03208* (2023).
- [6] Yinpeng Dong, Huanran Chen, Jiawei Chen, Zhengwei Fang, Xiao Yang, Yichi Zhang, Yu Tian, Hang Su, and Jun Zhu. 2023. How Robust is Google’s Bard to Adversarial Image Attacks? *arXiv preprint arXiv:2309.11751* (2023).
- [7] Zeming Dong, Qiang Hu, Yuejun Guo, Maxime Cordy, Mike Papadakis, Zhenya Zhang, Yves Le Traon, and Jianjun Zhao. 2023. Mixcode: Enhancing code classification by mixup-based data augmentation. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 379–390.
- [8] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 865–865.
- [9] Yang Feng, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. 2020. DeepGini: prioritizing massive tests to enhance the robustness of deep neural networks (*ISSTA 2020*). Association for Computing Machinery, New York, NY, USA, 177–188. <https://doi.org/10.1145/3395363.3397357>
- [10] Yarin Gal and Zoubin Ghahramani. 2016. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*. PMLR, 1050–1059.
- [11] Xinyu Gao, Yang Feng, Yining Yin, Zixi Liu, Zhenyu Chen, and Baowen Xu. 2022. Adaptive test selection for deep neural networks (*ICSE ’22*). Association for Computing Machinery, New York, NY, USA, 73–85. <https://doi.org/10.1145/3510003.3510232>
- [12] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. 2017. On calibration of modern neural networks. In *International conference on machine learning*. PMLR, 1321–1330.
- [13] Neil Houlsby, Ferenc Huszár, Zoubin Ghahramani, and Máté Lengyel. 2011. Bayesian active learning for classification and preference learning. *arXiv preprint arXiv:1112.5745* (2011).
- [14] Qiang Hu, Yuejun Guo, Xiaofei Xie, Maxime Cordy, Lei Ma, Mike Papadakis, and Yves Le Traon. 2024. Test Optimization in DNN Testing: A Survey. *ACM Trans. Softw. Eng. Methodol.* (jan 2024). <https://doi.org/10.1145/3643678>
- [15] Qiang Hu, Yuejun Guo, Xiaofei Xie, Maxime Cordy, Wei Ma, Mike Papadakis, and Yves Le Traon. 2023. Evaluating the robustness of test selection methods for deep neural networks. *arXiv preprint arXiv:2308.01314* (2023).
- [16] Yuheng Huang, Jiayang Song, Zhijie Wang, Huaming Chen, and Lei Ma. 2023. Look before you leap: An exploratory study of uncertainty measurement for large language models. *arXiv preprint arXiv:2307.10236* (2023).
- [17] Lorenz Kuhn, Yarin Gal, and Sebastian Farquhar. 2023. Semantic uncertainty: Linguistic invariances for uncertainty estimation in natural language generation. *arXiv preprint arXiv:2302.09664* (2023).
- [18] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
- [19] YU LI, Min LI, Qiuxia LAI, Yunnan Liu, and Qiang Xu. 2021. TestRank: Bringing Order into Unlabeled Test Instances for Deep Learning Tasks. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 20874–20886. https://proceedings.neurips.cc/paper_files/paper/2021/file/ae78510109d46b0a6eef9820a4ca95d6-Paper.pdf
- [20] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. CCTest: Testing and Repairing Code Completion Systems. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) (*ICSE ’23*). IEEE Press, 1238–1250. <https://doi.org/10.1109/ICSE48619.2023.00110>
- [21] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).
- [22] Wei Ma, Shangqing Liu, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu. 2023. The Scope of ChatGPT in Software Engineering: A Thorough Investigation. *arXiv preprint arXiv:2305.12138* (2023).
- [23] Wei Ma, Mike Papadakis, Anestis Tsakmalis, Maxime Cordy, and Yves Le Traon. 2021. Test selection for deep learning systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–22.

- [24] Vukosi Marivate and Tshephisho Sefara. 2020. Improving short text classification through global augmentation methods. In *International Cross-Domain Conference for Machine Learning and Knowledge Extraction*. Springer, 385–399.
- [25] George A. Miller. 1995. WordNet: a lexical database for English. *Commun. ACM* 38, 11 (nov 1995), 39–41. <https://doi.org/10.1145/219717.219748>
- [26] Qiwei Peng, Yekun Chai, and Xuhong Li. 2024. HumanEval-XL: A Multilingual Code Generation Benchmark for Cross-lingual Natural Language Generalization. *arXiv preprint arXiv:2402.16694* (2024).
- [27] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.). <https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/a5bfc9e07964f8dddeb95fc584cd965d-Abstract-round2.html>
- [28] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
- [29] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 476–480. <https://doi.org/10.1109/ICSME.2014.77>
- [30] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. 2023. Alpaca: A strong, replicable instruction-following model. *Stanford Center for Research on Foundation Models*. <https://crfm.stanford.edu/2023/03/13/alpaca.html> 3, 6 (2023), 7.
- [31] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Roziere, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
- [32] Daniele Vitale, Paolo Ferragina, and Ugo Scaiella. 2012. Classification of Short Texts by Deploying Topical Annotations. In *Advances in Information Retrieval - 34th European Conference on IR Research, ECIR 2012, Barcelona, Spain, April 1-5, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7224)*, Ricardo Baeza-Yates, Arjen P. de Vries, Hugo Zaragoza, Berkant Barla Cambazoglu, Vanessa Murdock, Ronny Lempel, and Fabrizio Silvestri (Eds.). Springer, 376–387. https://doi.org/10.1007/978-3-642-28997-2_32
- [33] Zan Wang, Hanmo You, Junjie Chen, Yingyi Zhang, Xuyuan Dong, and Wenbin Zhang. 2021. Prioritizing Test Inputs for Deep Neural Networks via Mutation Analysis. In *Proceedings of the 43rd International Conference on Software Engineering (Madrid, Spain) (ICSE '21)*. IEEE Press, 397–409. <https://doi.org/10.1109/ICSE43902.2021.00046>
- [34] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.
- [35] Yining Yin, Yang Feng, Shihao Weng, Zixi Liu, Yuan Yao, Yichi Zhang, Zhihong Zhao, and Zhenyu Chen. 2023. Dynamic Data Fault Localization for Deep Neural Networks. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1345–1357.
- [36] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating instruction-tuned large language models on code comprehension and generation. *arXiv preprint arXiv:2308.01240* (2023).
- [37] Wenxuan Zhang, Yue Deng, Bing Liu, Sinno Jialin Pan, and Lidong Bing. 2023. Sentiment Analysis in the Era of Large Language Models: A Reality Check. *arXiv preprint arXiv:2305.15005* (2023).
- [38] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2023. Siren’s song in the AI ocean: a survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219* (2023).
- [39] Haibin Zheng, Jinyin Chen, and Haibo Jin. 2023. CertPri: certifiable prioritization for deep neural networks via movement cost in feature space. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1–13.
- [40] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910* (2022).
- [41] Fei Zhu, Zhen Cheng, Xu-Yao Zhang, and Cheng-Lin Liu. 2023. Openmix: Exploring outlier samples for misclassification detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 12074–12083.
- [42] Fei Zhu, Zhen Cheng, Xu-Yao Zhang, and Cheng-Lin Liu. 2023. Rethinking confidence calibration for failure prediction. *arXiv preprint arXiv:2303.02970* (2023).

A RELIABILITY DIAGRAMS FOR LLMs

Figure 4 and Figure 5 depict the reliability diagrams of LLM with and without using MuCS.

Fig. 4. Reliability diagrams of LLMs.

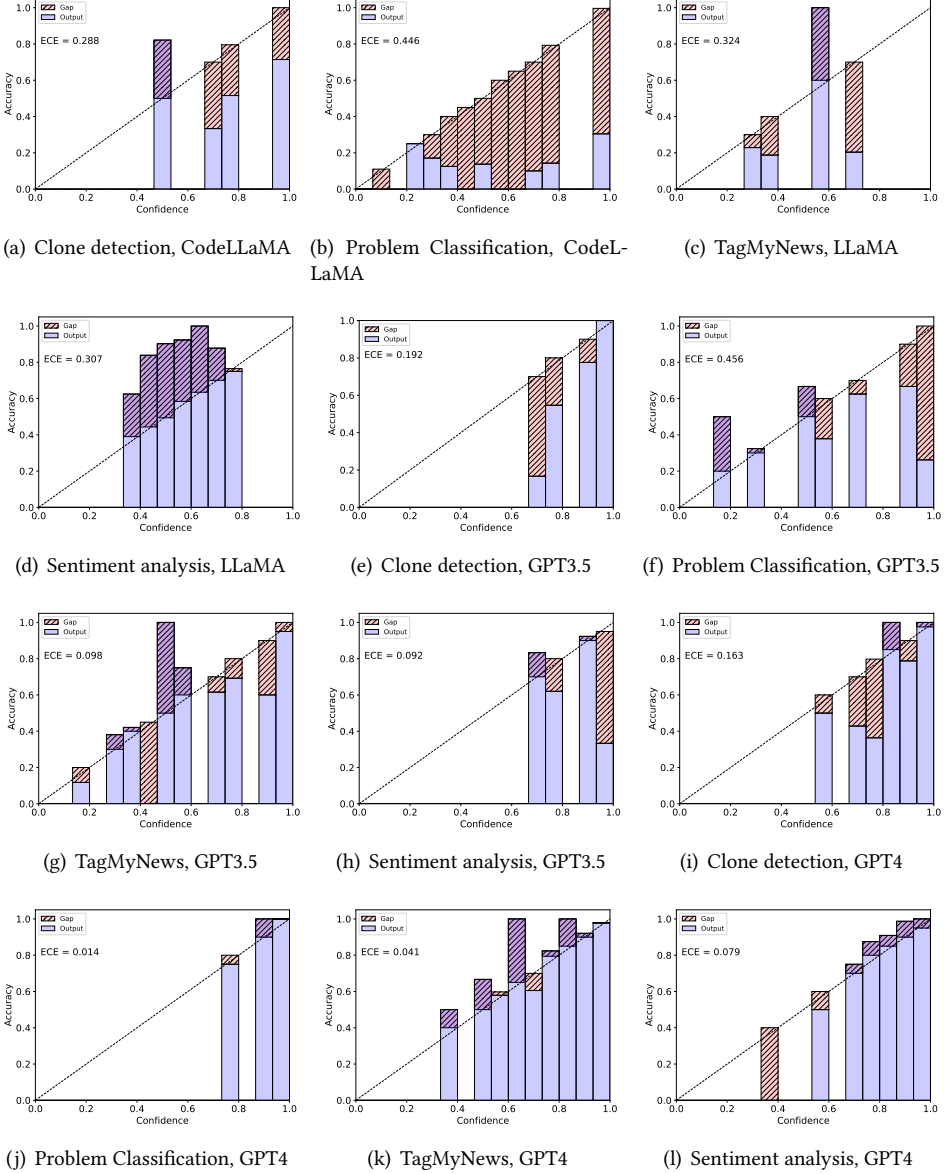


Fig. 5. Reliability diagrams of LLMs after using MuCS.

