Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE - EMPIRICAL & PRACTICE

Evaluation and Improvement of Test Selection for Large Language Models

Lili Quan*[1] | Jin Wen*[2] | Qiang Hu[1] | Maxime Cordy[2] | Yuheng Huang[3] | Lei Ma[3,4] | Xiaohong Li[1]

[1]Tianjin University, Tianjin, China

[2]University of Luxembourg, Luxembourg

[3]The University of Tokyo, Tokyo, Japan

[4]University of Alberta, Alberta, Canada

Correspondence
Qiang Hu, School of Cyber Security, Tianjin University, Tianjin, China
Email: qianghu@tju.edu.cn

Abstract

Large language models (LLMs) have recently achieved significant success across various application domains, garnering substantial attention from different communities. Unfortunately, many *faults* still exist that LLM cannot properly predict. Such faults will harm the usability of LLMs in general and could introduce safety issues in reliability-critical systems such as autonomous driving systems. How to quickly reveal these faults in real-world datasets that LLM could face is important, but challenging. The major reason is that the ground truth is necessary but the data labeling process is heavy considering the time and human effort. To handle this problem, in the conventional deep learning testing field, test selection methods have been proposed for efficiently evaluating deep learning models by prioritizing faults. However, despite their importance, the usefulness of these methods on LLMs is unclear and underexplored. In this paper, we conduct the first empirical study to investigate the effectiveness of existing test selection methods for LLMs. Experimental results on four different tasks (including both code tasks and natural language processing tasks) and four LLMs (e.g., LLaMA3 and GPT4) demonstrated that simple methods such as Margin perform well on LLMs but there is still a big room for improvement. Based on the study, we further propose **MuCS**, a prompt **Mu**tation-based prediction **C**onfidence **S**moothing framework to boost the test selection capability. Concretely, multiple prompt mutation techniques have been proposed to help collect diverse outputs for confidence smoothing. The results show that our proposed framework significantly enhances existing methods with test relative coverage improvement by up to 70.53%.

KEYWORDS
Deep Learning Testing, LLMs, Test Selection

1 | INTRODUCTION

In recent 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 conventional methods in many areas, such as question answering[1], sentiment analysis[2], and code understanding[3] that make LLMs become the first choice of deep learning models in such tasks. Besides the most famous application ChatGPT , multiple LLMs have been proposed, e.g., LLaMA[4], Alpaca[5], and Cerebras-GPT[6]. 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[7] and CodeLLaMA[8]. 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[9]. Besides, recent work[10] showed that LLMs can be easily fooled by adversarial attacks and are not robust. Furthermore, considering SE tasks, existing code-related LLMs cannot

* These authors contributed equally to this work.
OpenAI, ChatGPT, https://chat.openai.com/

capture equivalent semantics when the given natural language prompt expresses the same meaning in different languages[11]. Those limitations remind us that LLMs are not always trustworthy, and it is necessary to carefully evaluate LLMs and quickly reveal the unreliable outputs of LLMs in practical usage.

The basic way to evaluate LLMs and reveal their weakness is to prepare test data as diverse as possible 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 tasks 0.03$ per 1,000 tokens of inputs for GPT4 using OpenAI API. Therefore, it is almost impossible to thoroughly evaluate LLMs on the target downstream task for people who have limited budgets. As a result, how to efficiently test LLMs and filter the mispredicted data (unreliable outputs) of LLMs with less effort becomes an urgent problem.

To tackle the data labeling issue mentioned above, in the field of deep learning testing, multiple test selection methods that quickly identify fault data (data the model has incorrect prediction) in the test set without using the labeling information[12] 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, such studied classical deep learning models are constructively different from LLMs, e.g., LLMs are pre-trained using a large amount of pre-training data with diverse tasks which makes their prediction confidence (most test selection methods rely on) unclear given a specific downstream task. Therefore, as no studies have explored the effectiveness of test selection methods for LLMs, it is unknown whether we can directly employ existing methods to evaluate LLMs and save our effort or not.

To bridge this gap, in this paper, we conduct the first study to investigate the effectiveness of existing test selection methods for LLMs. The major challenge in the study is that most test selection methods are proposed for classification tasks and use the output probabilities for data prioritization, but LLMs normally output a sequence of tokens without such a clear output probability for the given task. To solve this, we design the prompt to guide LLMs to produce the output with their confidence. For datasets that have more than two classes, we add examples (few-shot) in each class in the prompt as guidance to ask LLMs to predict new data. In total, our study covers nine test selection methods including both learning-based methods, e.g., TestRank[13], and output-based methods, e.g., ATS[14]. Based on the experiment results collected from four datasets (e.g., sentiment analysis and code clone detection) and four LLMs (e.g., LLaMA3 and GPT4). We found that 1) LLMs are not well-calibrated and overconfident in clone detection, problem classification, and news classification tasks, 2) simple methods such as Margin perform better compared to others, and 3) however, existing test selection methods perform relatively pool on LLMs compared their performance on classical deep learning models. There is a need to propose methods to further enhance existing test selection methods.

To do so, inspired by[15,16] which utilize mutation testing techniques to help test deep learning models, we introduce **MuCS**, a prompt **Mu**tation-based prediction **C**onfidence **S**moothing framework to enhance existing test selection methods. Specifically, we mutate the prompt by transformation 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 test selection using current test selection methods. We compare the effectiveness of test selection methods with and without our confidence smoothing method and found that using MuCS significantly enhances the performance of existing methods by up to 70.53%.

To summarize, the main contributions of this paper are:

- This is the first study that explores the effectiveness of test selection methods on LLMs. We released the implementation and used datasets[17] to facilitate further research in this direction.
- We found that 1) LLMs are overconfident in clone detection, problem classification, and news classification tasks; 2) the prediction confidence of LLMs is concentrated in a few intervals; and 3) simple methods perform relatively better than others.
- We propose MuCS, a prompt mutation-based method to smooth the confidence of LLMs and enhance the effectiveness of existing test selection 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 test selection for deep neural networks and large language model testing.

2.1 | Test Selection for Deep Neural Networks

To save the labeling budget, multiple test selection methods[18,19,20,21] have been proposed. A recent survey[12] reviewed test selection methods and divided them into test selection methods, sampling-based model retraining methods, model selection methods, and performance estimation methods. We focus on the test selection methods.

The early method DeepGini[22] has been proposed by Feng *et al.* to reveal inputs that are more likely been mispredicted by the model. DeepGini[22] 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, Wang *et al.*[16] 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.*[14] 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.*[23] first empirically proved that simple methods (i.e., output probability-based methods) perform well on test selection. 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.*[24] conducted a comprehensive empirical study to explore the potential of test selection methods and active learning methods on test selection. 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.*[25] 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.*[13] proposed TestRank which uses graph neural networks to learn the difference between benign inputs and faults for test selection.

Different from these works, our work is the first to explore the effectiveness of test selection methods for large language models. We found that existing methods cannot detect faults in LLMs well 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 has become one of the most important directions in almost all communities.

Chang *et al.* surveyed works that focus on evaluating LLMs[26]. 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.*[27] 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.*[28] 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.*[3] 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.*[29] 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.*[30] 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 generation 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.

Existing LLM testing works mainly focus on constructing new and challenging datasets (by manual data collection and adversarial data generation, etc.) to validate the capability of LLMs, which can be seen as test augmentation techniques. In contrast, our work targets test selection from the perspective of software testing for LLM testing.

3 | EMPIRICAL STUDY

Our study focuses on the problem of test selection of LLMs. We consider both learning-based test selection methods and output-probability-based methods in the literature. Based on the previous definition [12], given a LLM M, an unlabeled test set X_{test}, and a labeling budget *budget*, test selection is to select a subset $\widetilde{X}$ of X_{test} such that $\widetilde{X}_{budget} = \arg\min_{\{X_i \subseteq X_{test} \wedge |X_i| = budget\}} \varrho(M, X_i, Y_i)$ where Y_i are the labels corresponding to X_i, and $\varrho(\cdot)$ is the performance measurement function. We follow the guidance of previous works [12,24,31] to select nine test selection methods in our study. We exclude neuron coverage-based methods and surprise adequacy methods in our study since the parameter size of LLMs is too big (>7B) and the coverage and adequacy score are difficult to compute.

3.1 | Test Selection Methods

In total, we collect nine test selection 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, expressed as $\mathsf{Max}\left(p_y \mid y \in \mathcal{Y}\right)$. Data samples with lower MaxP scores are more likely to be faults.
- **DeepGini** defines a Gini score calculated by $Gini(x) = 1 - \sum_{y_i \in \mathcal{Y}} {p_{y_i}}^2(x)$ 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 test selection 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.

TABLE 1 Datasets and models used in our study.

Task	Class Number	Model	Test Size	Accuracy
Clone Detection (CD)	2	DeepSeekCoder	1000	69.4%
		GPT3.5	200	75.5%
		GPT4	200	77.0%
Problem Classification (PC)	5	DeepSeekCoder	1000	42.6%
		GPT3.5	200	36.0%
		GPT4	200	100.0%
Sentiment Analysis (Sentiment)	3	LLaMA3	1000	68.8%
		GPT3.5	150	82.0%
		GPT4	150	90.0%
TagMyNews (TMN)	7	LLaMA3	1400	46.7%
		GPT3.5	200	60.0%
		GPT4	200	82.0%

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 test selection.

3.2 | Datasets and Models

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. Due to the high cost of assessing close-source LLMs, we prepare relatively more test data for open-source LLMs than close-source LLMs.

Clone Detection (CD): Utilizing the BigCloneBench[32] dataset, we randomly select 1000 and 200 items to investigate the code clone detection ability of open-source LLMs and closed-source LLMs, respectively. 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. Note that, some test selection methods that require more than two classes such as ATS cannot apply to clone detection tasks.

Problem Classification (PC): Based on the Java250[33] dataset from the CodeNet Project, problem classification focuses on the classification of programming challenges. 1000 samples and 200 samples across 5 types of problems are randomly chosen for evaluating open-source and closed-source LLMs. This task measures 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. We employ the one-shot setting (one-shot examples are randomly selected from the original dataset) for this task due to the poor performance of zero-shot evaluation (nearly random guess).

Sentiment Analysis (Sentiment): For this task, 1000 samples and 150 samples are extracted from the Sentiment Analysis of IMDB Movie Reviews dataset to evaluate open-source and closed-source LLMs, respectively. The challenge for models is to predict the sentiment of movie reviews, negative, neutral, and positive, ranging from 0 to 1, evaluating the capacity of models for natural language understanding and emotional tone assessment.

TagMyNews (TMN): Using the TagMyNews Dataset, we randomly picked 1400 and 200 articles with seven categories, which are sourced from RSS feeds of popular newspapers[34] for open-source and closed-source LLMs, respectively. This task aims to assess the efficacy of the model in accurately categorizing news articles based on their textual content. Due to the same reason as the problem classification task, we employ the one-shot setting for this task.

LLaMA3 and DeepSeekCoder. LLaMA3[4] is known for its auto-regressive architecture aimed at chat applications. The parameters of the series of LLaMA3 models range from 8 billion to 70 billion and are learned from publicly available online data amounting to 2 trillion tokens. Besides, LLaMA3 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. DeepSeeker-Coder consists of a series of code language models, which achieve competitive performance on different code-related tasks[35]. DeepSeek-Coder not only beats most open-sourced LLMs but also surpasses existing advancing

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

close-source LLMs like Codex and GPT-3.5. We employ the instruct version (meta-llama/Meta-Llama-3-8B-Instruct and deepseek-ai/DeepSeek-Coder-V2-Lite-Instruct) available on the public Hugging Face platform for our experiments.

Prompt Template:

Given the following two code snippets, could you give me your prediction score on whether the two code snippets are semanticlly similar? Please give your predict probability score as a number between 0 and 1. If you think the two code snippets are definitely the same meaning, please give a score close to 1. If you think the two code snippets are not similar, please give a score close to 0. Remember to write your response strictly as the format, no extra analysis or sentence analysis please.

Format:
similar score: 0.5

Code 1: ```[Input_1]```
Code 2: ```[Input_2]```

FIGURE 1 Prompt template of clone detection.

GPT3.5 and GPT4[36]. GPT3.5 is a fined-tuned version of GPT3 developed in January 2022. It is built on Transformer architecture while emphasizing 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 test selection (RQ2) on them.

3.3 | Prompt Design

The quality of input prompts highly affects the output of LLMs. To obtain the output probabilities of LLMs given the input, we add guidance in the prompt to ask LLMs to produce their prediction confidence of each category. Figure 1 presents the template example of the code clone detection task. For multiple class classification tasks such as problem classification, we use one-shot examples to lead LLMs to produce more accurate prediction confidence, the detailed prompts can be found at our project site[17].

3.4 | Research Question

Our study aims to explore the effectiveness of existing test selection methods on LLMs. Concretely, we answer the following two research questions:

- **RQ1: How confident are LLMs in their prediction?** Most (seven out of nine) of the test selection methods are purely output uncertainty-based. Therefore, checking the prediction confidence (an important indicator to measure the output uncertainty) produced by LLMs before running test selection methods is necessary.
- **RQ2: How effective are test selection 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.

3.5 | Evaluation Metrics.

In RQ1, we follow the previous work[37] 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, $m \in \{1, \ldots, M\}$) 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:

$$ECE(X) = \sum_{m=1}^{M} \frac{|B_m|}{|X|} \left| Acc(B_m) - Avg\_Conf(B_m) \right|$$

In RQ2, we follow the previous work[13] and evaluate the effectiveness of test selection 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:

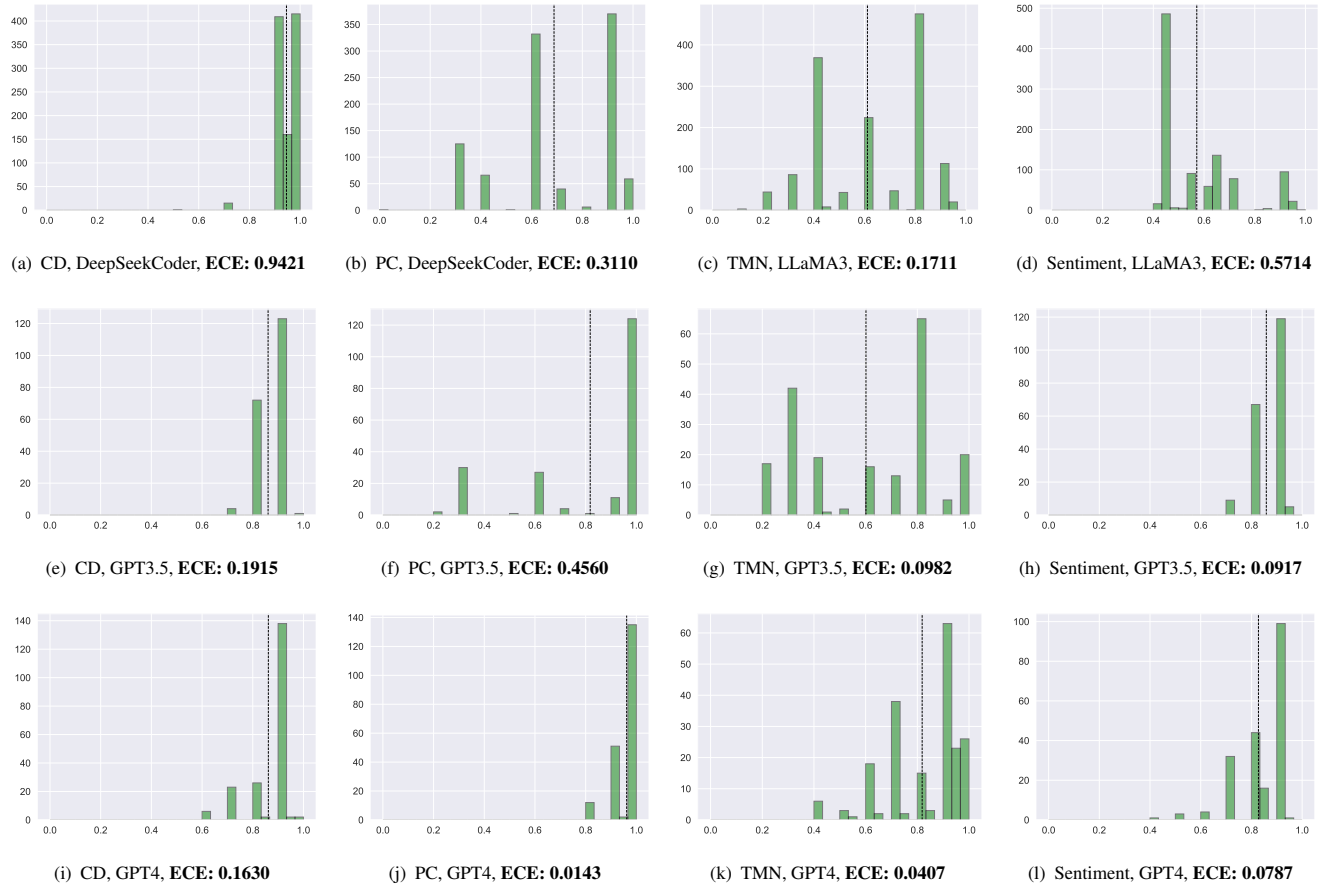
$$TRC(X) = \frac{|\widetilde{X}_{faults}|}{Min\left(budget, |X_{faults}|\right)}$$

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

3.6 | Configurations

Configuration of test selection 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 test selection. 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 from 10% to 90% with an interval of 10%.

Configuration of LLMs. In our experiments, we set *top_p=0.95* for open-sourced LLaMA3 and DeepSeek-Coder, and *top_p=1* for closure GPT3.5 and 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 LLaMA3 and DeepSeek-Coder 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.



(a) CD, DeepSeekCoder, **ECE: 0.9421** (b) PC, DeepSeekCoder, **ECE: 0.3110** (c) TMN, LLaMA3, **ECE: 0.1711** (d) Sentiment, LLaMA3, **ECE: 0.5714**

(e) CD, GPT3.5, **ECE: 0.1915** (f) PC, GPT3.5, **ECE: 0.4560** (g) TMN, GPT3.5, **ECE: 0.0982** (h) Sentiment, GPT3.5, **ECE: 0.0917**

(i) CD, GPT4, **ECE: 0.1630** (j) PC, GPT4, **ECE: 0.0143** (k) TMN, GPT4, **ECE: 0.0407** (l) Sentiment, GPT4, **ECE: 0.0787**

FIGURE 2 Distribution of prediction confidence and ECE for LLMs. The dashed line indicates the average confidence.

3.7 | Implementation and Environment

For the test selection 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 Face 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 high-performance computer cluster. Each cluster node runs 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, as most test selection methods rely on prediction confidence, we check whether LLMs can confidently predict the data in our studied tasks. Figure 2 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 is that GPT4 has the most confidence in its predictions compared to other LLMs. On average, the confidence of DeepSeekCoder, LLaMA3, GPT3.5, and GPT4 on these four datasets are 81.72%, 59.15%, 77.01%, and 87.27%, respectively. Comparing the accuracy of LLMs and their average confidence (which is a notion of miscalibration[37]), we found that our LLMs are not well-calibrated to these four tasks. For example, for DeepSeekCoder on the clone detection task, the accuracy and average confidence are 69.4% and 94.61%,

Hugging Face, https://huggingface.co/

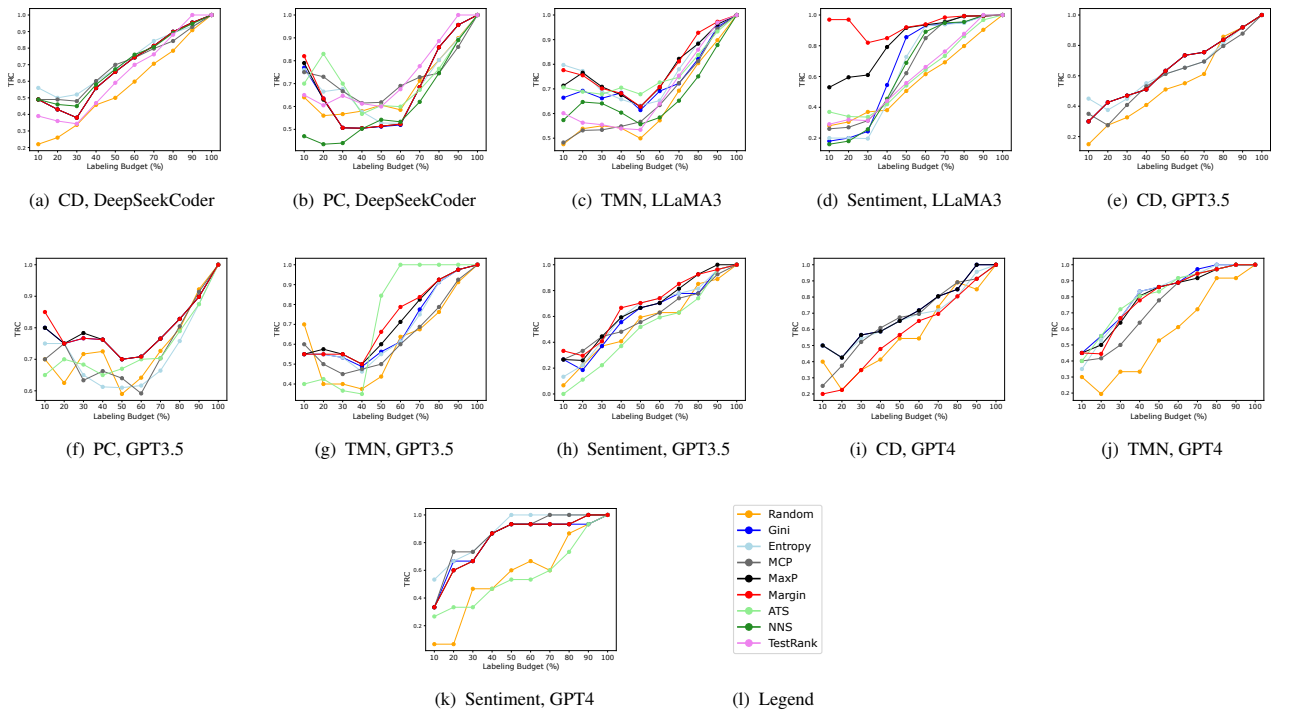
respectively. This means DeepSeekCoder is overconfident with its prediction even though it has poor performance. Overall, except for TagMyNews, LLMs are overconfident in the other three tasks with 16.78% higher confidence than the true accuracy. Additionally, the results show that prediction confidence 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 test selection methods (e.g., MaxP) could produce similar uncertainty scores to these inputs.

Interestingly, 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. Investigating the exact reason behind this could be future work to better understand what has been learned by the pre-trained LLMs. As there are no faults in GPT4 for the problem classification task, we exclude this model from the following test selection 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 test selection

In this RQ, we explore the performance of existing test selection methods for LLMs. As mentioned in Section 3.2, we exclude models that have perfect performance (100% accuracy) on the studied tasks.



(a) CD, DeepSeekCoder (b) PC, DeepSeekCoder (c) TMN, LLaMA3 (d) Sentiment, LLaMA3 (e) CD, GPT3.5

(f) PC, GPT3.5 (g) TMN, GPT3.5 (h) Sentiment, GPT3.5 (i) CD, GPT4 (j) TMN, GPT4

(k) Sentiment, GPT4 (l) Legend

FIGURE 3 Test relative coverage.

Figure 3 depicts the TRC values achieved by test selection methods in each labeling budget, and Table 2 summarizes the averaged TRC values across all considered labeling budgets. First, considering the averaged results, we can see that all methods have relatively better results than random selection, which indicates the potential of their usefulness. However, we found that the best averaged TRC value on LLMs is only 0.7321, which is far away from the idea detection performance, TRC = 1. Worsely, in some cases, e.g., Sentiment-LLaMA3, some methods have less than 0.2 TRC score, which never happened in conventional

TABLE 2 Average test relative coverage of each method. The best method is highlighted in green.

	Random	Gini	Entropy	Mcp	Maxp	Margin	ATS	NNS	TestRank
DeepSeekCoder-CD	0.5301	0.6586	0.6973	0.6746	0.6586	0.6586	-	0.6732	0.6106
GPT3.5-CD	0.5121	0.6202	0.6132	0.5774	0.6202	0.6202	-	-	-
GPT4-CD	0.5501	0.6777	0.6535	0.6371	0.5424	0.6777	-	-	-
DeepSeekCoder-PC	0.6614	0.6609	0.6796	0.7116	0.6631	0.6670	0.7031	0.5754	0.7169
GPT3.5-PC	0.7160	0.7755	0.6984	0.7110	0.7774	0.7811	0.7134	-	-
LLaMA3-TMN	0.6191	0.7227	0.7569	0.6419	0.7627	0.7737	0.7446	0.6541	0.6683
GPT3.5-TMN	0.5889	0.6620	0.6551	0.6139	0.6903	0.7042	0.7096	-	-
GPT4-TMN	0.5395	0.8031	0.7889	0.7265	0.7815	0.7784	0.7944	-	-
LLaMA3-Sentiment	0.5392	0.6490	0.6176	0.6332	0.8131	0.9377	0.5796	0.6147	0.5806
GPT3.5-Sentiment	0.5177	0.5852	0.5951	0.5728	0.6305	0.6543	0.4609	-	-
GPT4-Sentiment	0.5259	0.8000	0.8667	0.8370	0.8000	0.8000	0.5259	-	-
Average	0.5727	0.6923	0.6929	0.6670	0.7036	0.7321	0.6540	0.6294	0.6441

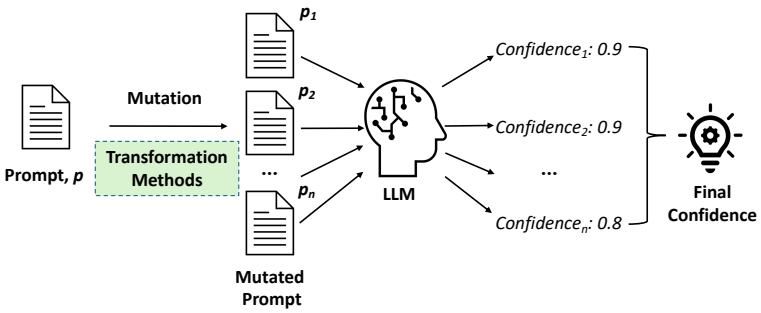


FIGURE 4 Workflow of MuCS.

deep learning models[13]. This means test selection for LLMs is a more challenging problem compared to test selection for conventional deep learning models.

Then, comparing each method, interestingly, we found that the simple method, Margin, which just utilizes the top-1 and top-2 output probabilities to measure the uncertainty, performs the best among our considered methods. This phenomenon is similar to the findings by existing works[38] which stated that simple methods perform the best for neural network test prioritization. Specifically, in more than half of cases (6 out of 11 cases), Margin achieved the highest averaged TRC scores. Methods that require embedding information for the data selection (NNS and TestRank) cannot stand out in LLMs. The reason could be that the embeddings extracted by LLMs are too high-dimensional and diverse to be learned by simple clustering methods (used by NNS) and graph neural networks (used by TestRank).

Answer to RQ2: Under our studied objects, the simple method, Margin, performs the best in test selection for LLMs. However, there is still a big room for improvement, and more effective methods are needed.

5 | MUCS: MUTATION-BASED CONFIDENCE SMOOTHING

As analyzed in Section 4.1, the concentrated confidence issue can harm the performance of output-based test selection methods (those methods will give similar uncertainty scores to the inputs), we need to diversify the prediction confidence of LLMs to enhance test selection. 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[16,15] 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 framework to enhance existing test selection methods by smoothing the prediction confidence from the input-level using prompt mutation.

Algorithm 1 MuCS

```
1: Input:
   • p: input prompt
   • OPs: mutation operators
   • M: LLM
   • K: perturbation size
   • n: number of generated mutants
2: Output: smoothed confidence C_smooth
3: Cs ← []
4: for i = 0 → n do                                          ▷ mutants generation
5:    for j = 0 → K do
6:       OP ← RandomSelection(OPs)
7:       p' ← OP(p)
8:       p ← p'
9:    end for                                                ▷ confidence calculation using Definition 1
10:   Cs.append(Confidence(M(p)))
11: end for                                                  ▷ confidence averaging
12: C_smooth ← Mean(Cs)
13: return Cs
```

Figure 4 presents the overall workflow of our proposed framework. 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, ..., p_n. For NLP tasks, we use text augmentation methods[39] (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[40] (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 $Confidence_1$, $Confidence_2$,..., $Confidence_n$ 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. MuCS first randomly selects multiple (defined by the perturbation size K) mutation operators OP (line 6) to mutate p and then generates p' (lines 7, 8). Then, MuCS collects all the prediction confidence produced by LLMs using all mutants p' (line 10). Finally, the averaged confidence is returned as the smoothed confidence (lines 12 and 13).

5.1 | Mutation Operators

Two types of mutation operators have been included in the framework, 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[41]. We use the default setting of TextAugment[39], $n = 1$ 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[42], $T = 0.01$ 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.

TABLE 3 Averaged test relative coverage (relative improvement compared to results without using MuSC in Table 2) cross all labeling budgets of each method after prompt mutation-based confidence smoothing.

	Gini-M	Entropy-M	MCP-M	MaxP-M	Margin-M	ATS-M	NNS-M	TestRank-M	BALD
DeepSeekCoder-CD	0.6886 (↑4.55)	0.6890 (↓1.20)	0.6592 (↓2.28)	0.6886 (↑4.55)	0.6886 (↑4.55)	-	0.6576 (↓2.32)	0.6134 (↑0.46)	0.7067
GPT3.5-CD	0.6873 (↑10.81)	0.5989 (↓2.33)	0.6369 (↑10.31)	0.6873 (↑10.81)	0.6873 (↑10.81)	-	-	-	0.5827
GPT4-CD	0.7550 (↑11.41)	0.7211 (↑10.34)	0.7318 (↑14.87)	0.7550 (↑39.20)	0.5107 (↓24.64)	-	-	-	0.4380
DeepSeekCoder-PC	0.7462 (↑12.91)	0.7134 (↑4.97)	0.6883 (↓3.27)	0.7412 (↑11.77)	0.6516 (↓2.31)	0.7212 (↑2.57)	0.6984 (↑21.37)	0.7425 (↑3.57)	0.7145
GPT3.5-PC	0.8344 (↑7.59)	0.8346 (↑19.49)	0.7116 (↑0.08)	0.8492 (↑9.24)	0.8343 (↑6.81)	0.7930 (↑11.16)	-	-	0.8389
LLaMA3-TMN	0.7637 (↑5.68)	0.7722 (↑2.02)	0.7073 (↑10.18)	0.7702 (↑0.99)	0.7653 (↓1.09)	0.7162 (↓3.82)	0.7156 (↑9.41)	0.7086 (↑6.03)	0.7434
GPT3.5-TMN	0.7431 (↑12.24)	0.7319 (↑11.72)	0.6208 (↑1.13)	0.7630 (↑10.54)	0.7481 (↑6.24)	0.7708 (↑8.63)	-	-	0.7407
GPT4-TMN	0.7691 (↓4.23)	0.7327 (↓7.12)	0.7451 (↑2.55)	0.7790 (↓0.32)	0.7821 (↑0.48)	0.7852 (↓1.16)	-	-	0.7037
LLaMA3-Sentiment	0.6417 (↓1.13)	0.5899 (↓4.48)	0.8423 (↑33.02)	0.7587 (↓6.70)	0.8643 (↓7.83)	0.6211 (↑7.16)	0.6006 (↓2.30)	0.6065 (↑4.46)	0.6199
GPT3.5-Sentiment	0.7654 (↑30.80)	0.7646 (↑28.49)	0.7959 (↑38.94)	0.7539 (↑19.58)	0.6782 (↑3.65)	0.7860 (↑70.53)	-	-	0.6379
GPT4-Sentiment	0.8593 (↑7.41)	0.8593 (↓0.85)	0.8444 (↑0.88)	0.8593 (↑7.41)	0.8519 (↑6.49)	0.7481 (↑42.24)	-	-	0.6667
Average	0.7503 (↑8.39)	0.7280 (↑5.05)	0.7258 (↑8.81)	0.7641 (↑8.60)	0.7329 (↑0.12)	0.7427 (↑13.57)	0.6681 (↑6.15)	0.6678 (↑3.67)	0.6720

- **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 investigate whether our proposed framework MuCS can help test selection for LLMs, we first analyze the effectiveness of existing test selection with MuCS as a plug-in. Then, we explore the reason why MuCS enhances test selection by analyzing its influences on the prediction confidence of LLMs. Our exploration plans to answer the following research questions.

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

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.

6.3 | RQ3: Effectiveness of MuCS

First, we explore whether the MuCS can enhance the existing test selection methods or not. Before that, since the input mutation can produce multiple predictions from LLMs which is similar to the dropout uncertainty [43]. We introduce another test selection method baseline which is based on dropout uncertainty, Bayesian Active Learning by Disagreement (BALD) [44].

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 $1 - \frac{count(mode(y^1,...,y^T))}{T}$, 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 Statistical analysis for comparing test selection methods with and without using MuCS.

		Gini	Entropy	Mcp	Maxp	Margin	ATS	NNS	TestRank
Welch's t-test	Significance	2.8873	1.7373	3.0355	2.5787	1.1567	3.1049	1.1405	0.7018
	P-value	4.10E-03	8.31E-02	2.56E-03	1.03E-02	2.48E-01	2.11E-03	2.56E-01	4.84E-01
Wilcoxon signed-rank test	Significance	500.5	1136.5	770.5	663.5	1273.5	283	148.5	42.5
	P-value	4.82E-10	2.47E-04	1.30E-06	9.15E-09	1.18E-03	1.95E-06	1.09E-02	2.11E-05

TABLE 5 ECE and variance of prediction distribution before and after confidence smoothing.

	ECE (↓)		Diversity (↓)	
Model	**Before**	**After**	**Before**	**After**
CD-DeepSeekCoder	0.9421	0.9112	11066.62	9535.22
CD-GPT35	0.1915	0.2563	584.96	655.29
CD-GPT4	0.1630	0.1638	696.62	542.42
PC-DeepSeekCoder	0.3110	0.1713	7963.02	1189.08
PC-GPT35	0.4560	0.3449	205.36	24.69
PC-GPT4	0.0143	-	332.00	-
TMN-LLaMA3	0.1711	0.0701	12338.18	1641.37
TMN-GPT35	0.0982	0.1030	205.36	24.69
TMN-GPT4	0.0407	0.0846	196.56	62.29
Sentiment-LLaMA3	0.5714	0.2345	8301.62	4898.62
Sentiment-GPT35	0.0917	0.1180	335.87	104.87
Sentiment-GP4	0.0787	0.1093	230.60	93.33
Average change	↑ **17.60%**		↑ **55.44%**	

Table 3 summarizes the average TRC scores achieved by different test selection methods across all considered labeling budgets using MuCS (with -M as a suffix), and the relative difference between the scores with and without using MuCS. The results clearly demonstrate that the MuCS significantly boosts the performance of test selection. Specifically, in 52 out of 71 cases, the performance of existing test selection methods has been increased with TRC improvements ranging from 0.48% to 70.58%. Considering the average results (the last row in the Table), MuCS can enhance all test selection methods with improvement by up to 13.57%. Besides, we conducted the statistical analysis using two methods Welch's t-test and Wilcoxon signed-rank test to investigate the significance of improvements introduced by MuCS. Concretely, we collect all TRC scores resulting from each method with and without using MuCS and compute the significance. The results are shown in Table 4. The t-test suggests that for four methods (Gini, MCP, MaxP, and ATS), the improvements are significant, and the signed-rank test demonstrates all improvements are significant (with P-value < 0.03). Therefore, we can conclude that MuCS is effective in boosting the existing test selection methods on LLMs.

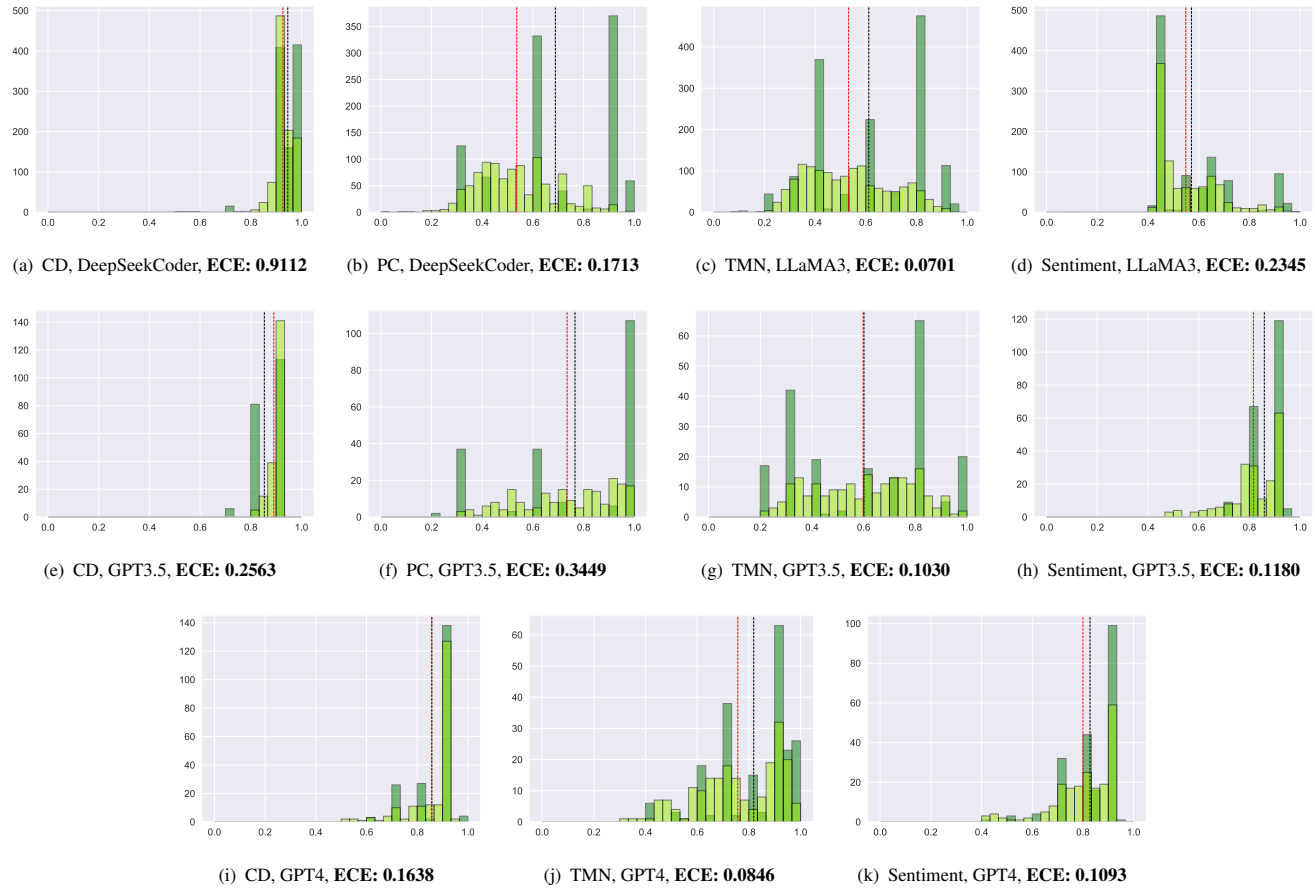
Comparing the improvements across different LLMs, we found that improvements in closed-source LLMs are greater than improvements in open-source LLMs. On average, the improvements in DeepSeekCoder, LLaMA3, GPT3.5, and GPT4 are 3.82%, 3.22%, 16.38%, and 7.87%, respectively. This means MuCS is an effective framework to enhance the black-box testing of LLMs. Then, after comparing each method after using MuCS, we found that MaxP, another simple method that directly utilizes the top-output probability as the indicator for data prioritization performs the best. Combining the conclusion in Section 4.2, simple methods perform better than learning-based methods with and without using MuCS.

Finally, even though the mutation-based confidence smooth can enhance the test selection for LLMs, the best method only has a 0.7641 average TRC score. 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 test selection methods by up to 70.53% 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 5 summarizes the average ECE scores. The results show that even though MuCS can reduce the average ECE scores (i.e., better calibration), there are still



(a) CD, DeepSeekCoder, **ECE: 0.9112** (b) PC, DeepSeekCoder, **ECE: 0.1713** (c) TMN, LLaMA3, **ECE: 0.0701** (d) Sentiment, LLaMA3, **ECE: 0.2345**

(e) CD, GPT3.5, **ECE: 0.2563** (f) PC, GPT3.5, **ECE: 0.3449** (g) TMN, GPT3.5, **ECE: 0.1030** (h) Sentiment, GPT3.5, **ECE: 0.1180**

(i) CD, GPT4, **ECE: 0.1638** (j) TMN, GPT4, **ECE: 0.0846** (k) Sentiment, GPT4, **ECE: 0.1093**

FIGURE 5 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.

multiple pairs that have similar ECE scores, e.g., Sentiment-GPT35 and Sentiment-GPT4. Moreover, previous works[45] stated the effectiveness of test selection methods is not directly related to the calibration of the model (most confidence calibration methods are useless or harmful for test selection). It is not clear whether the improvements are introduced by better calibration.

Then, we check if MuCS can increase the diversity of confidence distribution to solve the confidence concentrated issue. We compute the variance of the number of confidence scores in each range to check this diversity. The results presented in Table 5 demonstrated that, except for one case CD-GPT35, LLMs have more diverse predictions on the datasets with an average 55.44% diversity improvement. Taking Figure 5(f) 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 for this task.

Answer to RQ4: MuCS significantly increases the diversity of confidence distribution of LLMs with an average 55.44% improvement and eliminates the concentrated confidence problem.

7 | DISCUSSION

7.1 | Confidence vs. Correctness

As mentioned before, all considered test selection methods rely on the assumption – there is a connection between the prediction confidence and the prediction correctness. However, it is unclear whether the assumption stands or not, especially for LLMs.

TABLE 6 Correlation between prediction confidence and prediction correctness.

	Without MuCS		With MuCS	
Model	**Significance**	**P-value**	**Significance**	**P-value**
CD-DeepSeekCoder	0.2753	7.43E-19	0.2243	1.09E-12
CD-GPT3.5	0.2636	1.21E-13	0.3032	9.41E-18
CD-GPT4	0.2969	4.71E-17	0.2823	1.70E-15
PC-DeepSeekCoder	-0.0522	9.92E-02	0.2402	1.38E-14
PC-GPT3.5	0.0952	5.22E-02	0.2546	1.07E-07
TMN-LLaMA3	0.1110	2.53E-05	0.2426	1.21E-20
TMN-GPT3.5	0.2904	2.97E-29	0.3238	2.47E-36
TMN-GPT4	0.3909	1.55E-53	0.4059	5.97E-58
Sentiment-LLaMA3	0.4603	1.34E-53	0.3730	2.33E-34
Sentiment-GPT3.5	0.2007	4.37E-03	0.2124	2.53E-03
Sentiment-GPT4	0.4263	3.10E-10	0.4332	1.49E-10

We design an exploratory study to answer this problem. Specifically, we label the corrected prediction as 1 and the incorrect prediction as 0 and then utilize Pearson correlation coefficient to quantify the relationship between confidence and correctness as shown in Table 6. We can see that there is a clear positive correlation between these two indicators except for the problem classification task before using MuCS. Moreover, after employing MuCS, all correlations become positive with a significance less than 0.03. This exploratory study demonstrates the relationship between confidence and correctness exists, and can be enhanced by MuCS.

7.2 | Limitation and Future Direction

Limitation. 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. However, from the view of executing time, the increased cost is negligible since it is possible to perform the prediction of a batch of data at the same time. There is indeed a trade-off between the monetary cost of accessing close-source LLMs and data labeling. This should be considered case by case.

Future direction. In this work, we only consider classification tasks since most existing test selection methods are proposed for these tasks. Some methods, such as PRIMA [16] can be used for none classification tasks directly. Unfortunately, applying it to LLMs is not practical as it needs to mutate deep learning models but LLMs are much larger than classical deep learning models. However, the most important tasks people would use are generation tasks, e.g., chatting using ChatGPT. One of the important directions in this field is to propose test selection methods for non-classification tasks, especially generation tasks, and then use these methods to help reveal critical faults in LLMs. Some metrics [46,47] have been designed for measuring the uncertainty in LLMs, how to combine these uncertainty metrics with existing methods could be the first try.

7.3 | Threats to Validity

The internal threat lies in the implementation of test selection methods, LLMs, and prompt mutation methods. The implementation of each test selection method comes from the official project provided in the 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 [39]. The implementation of code refactoring methods modified from the previous work [42].

The external threat comes from our studied tasks, datasets, LLMs, and test selection methods. As mentioned in Section 7.2, 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 accuracy analysis of LLMs. For the test selection methods, we follow the previous works [12,24,31] and explore 10 test selection 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 reproducing results reported in our work and for future research.

8 | CONCLUSION

In this work, we conducted the first empirical study to explore the potential of test selection methods for LLMs. Based on the experiments on four tasks including NLP and code tasks, four LLMs, and nine test selection methods, we found LLMs are overconfident with their predictions and simple methods such as Margin performs the best 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 test selection by up to 70.53% in terms of test relative coverage score.

AUTHOR CONTRIBUTIONS

Lili Quan, Jin Wen, and Qiang Hu primarily conducted the experiments. Maxime Cordy, Yuheng Huang, Lei Ma, and Xiaohong Li contributed to discussions and manuscript revisions. All authors read and approved the final manuscript.

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China (2023YFB3107100) and the University of Luxembourg under grant number R-STR-5009-00-B.

FINANCIAL DISCLOSURE

None reported.

CONFLICT OF INTEREST

The authors declare that they have no potential conflict of interest.

DATA AVAILABILITY STATEMENT

The code and data used in this study are publicly available at the following repository: https://anonymous.4open.science/r/LLM_FD-64DA/.

References

1. Zhou Y, Muresanu AI, Han Z, et al. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910.* 2022.
2. Zhang W, Deng Y, Liu B, Pan SJ, Bing L. Sentiment Analysis in the Era of Large Language Models: A Reality Check. *arXiv preprint arXiv:2305.15005.* 2023.
3. Ma W, Liu S, Wang W, et al. The Scope of ChatGPT in Software Engineering: A Thorough Investigation. *arXiv preprint arXiv:2305.12138.* 2023.
4. Touvron H, Lavril T, Izacard G, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971.* 2023.
5. Taori R, Gulrajani I, Zhang T, et al. Alpaca: A strong, replicable instruction-following model. *Stanford Center for Research on Foundation Models. https://crfm. stanford. edu/2023/03/13/alpaca. html.* 2023;3(6):7.
6. Dey N, Gosal G, Khachane H, et al. Cerebras-gpt: Open compute-optimal language models trained on the cerebras wafer-scale cluster. *arXiv preprint arXiv:2304.03208.* 2023.
7. Li R, Allal LB, Zi Y, et al. StarCoder: may the source be with you!. *arXiv preprint arXiv:2305.06161.* 2023.
8. Roziere B, Gehring J, Gloeckle F, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950.* 2023.
9. Zhang Y, Li Y, Cui L, et al. Siren's song in the AI ocean: a survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219.* 2023.
10. Dong Y, Chen H, Chen J, et al. How Robust is Google's Bard to Adversarial Image Attacks?. *arXiv preprint arXiv:2309.11751.* 2023.
11. Peng Q, Chai Y, Li X. HumanEval-XL: A Multilingual Code Generation Benchmark for Cross-lingual Natural Language Generalization. *arXiv preprint arXiv:2402.16694.* 2024.
12. Hu Q, Guo Y, Xie X, et al. Test Optimization in DNN Testing: A Survey. *ACM Trans. Softw. Eng. Methodol.*. 2024. doi: 10.1145/3643678
13. Li Y, Li M, Lai Q, Liu Y, Xu Q. Testrank: Bringing order into unlabeled test instances for deep learning tasks. *Advances in Neural Information Processing Systems.* 2021;34:20874–20886.
14. Gao X, Feng Y, Yin Y, Liu Z, Chen Z, Xu B. Adaptive test selection for deep neural networks. 2022:73–85. doi: 10.1145/3510003.3510232
15. Li Z, Wang C, Liu Z, et al. Cctest: Testing and repairing code completion systems. In: IEEE. 2023:1238–1250.
16. Wang Z, You H, Chen J, Zhang Y, Dong X, Zhang W. Prioritizing test inputs for deep neural networks via mutation analysis. In: IEEE. 2021:397–409.
17. MuCS.; 2024.
18. Al-Qadasi H, Falcone Y, Bensalem S. DeepAbstraction++: Enhancing Test Prioritization Performance via Combined Parameterized Boxes. In: Springer. 2023:77–93.
19. Zheng H, Chen J, Jin H. CertPri: certifiable prioritization for deep neural networks via movement cost in feature space. In: IEEE. 2023:1–13.
20. Zhu F, Cheng Z, Zhang XY, Liu CL. Openmix: Exploring outlier samples for misclassification detection. In: 2023:12074–12083.
21. Yin Y, Feng Y, Weng S, et al. Dynamic Data Fault Localization for Deep Neural Networks. In: 2023:1345–1357.
22. Feng Y, Shi Q, Gao X, Wan J, Fang C, Chen Z. DeepGini: prioritizing massive tests to enhance the robustness of deep neural networks. In: ISSTA 2020. Association for Computing Machinery 2020; New York, NY, USA:177–188
23. Bao S, Sha C, Chen B, Peng X, Zhao W. In Defense of Simple Techniques for Neural Network Test Case Selection. In: ISSTA 2023. Association for Computing Machinery 2023; New York, NY, USA:501–513
24. Ma W, Papadakis M, Tsakmalis A, Cordy M, Traon YL. Test selection for deep learning systems. *ACM Transactions on Software Engineering and Methodology (TOSEM).* 2021;30(2):1–22.
25. Hendrycks D, Gimpel K. A baseline for detecting misclassified and out-of-distribution examples in neural networks. *arXiv preprint arXiv:1610.02136.* 2016.
26. Chang Y, Wang X, Wang J, et al. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology.* 2023.
27. Xu FF, Alon U, Neubig G, Hellendoorn VJ. A systematic evaluation of large language models of code. In: 2022:1–10.
28. Liu J, Xia CS, Wang Y, Zhang L. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems.* 2024;36.
29. Yuan Z, Liu J, Zi Q, Liu M, Peng X, Lou Y. Evaluating instruction-tuned large language models on code comprehension and generation. *arXiv preprint arXiv:2308.01240.* 2023.
30. Du X, Liu M, Wang K, et al. Evaluating Large Language Models in Class-Level Code Generation. In: IEEE Computer Society. 2024:865–865.

31. Hu Q, Guo Y, Xie X, et al. Evaluating the robustness of test selection methods for deep neural networks. *arXiv preprint arXiv:2308.01314.* 2023.

32. Svajlenko J, Islam JF, Keivanloo I, Roy CK, Mia MM. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In: IEEE Computer Society 2014:476–480

33. Puri R, Kung DS, Janssen G, et al. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In: Vanschoren J, Yeung S., eds. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual* 2021.

34. Vitale D, Ferragina P, Scaiella U. Classification of Short Texts by Deploying Topical Annotations. In: Baeza-Yates R, Vries dAP, Zaragoza H, et al. , eds. *Advances in Information Retrieval - 34th European Conference on IR Research, ECIR 2012, Barcelona, Spain, April 1-5, 2012. Proceedings*. 7224 of *Lecture Notes in Computer Science*. Springer 2012:376–387

35. Guo D, Zhu Q, Yang D, et al. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. *CoRR.* 2024;abs/2401.14196. doi: 10.48550/ARXIV.2401.14196

36. Achiam J, Adler S, Agarwal S, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774.* 2023.

37. Guo C, Pleiss G, Sun Y, Weinberger KQ. On calibration of modern neural networks. In: PMLR. 2017:1321–1330.

38. Weiss M, Tonella P. Simple techniques work surprisingly well for neural network test prioritization and active learning (replicability study). In: 2022:139–150.

39. Marivate V, Sefara T. Improving short text classification through global augmentation methods. In: Springer. 2020:385–399.

40. Al Dallal J, Abdin A. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering.* 2017;44(1):44–69.

41. Miller GA. WordNet: a lexical database for English. *Commun. ACM.* 1995;38(11):39–41. doi: 10.1145/219717.219748

42. Dong Z, Hu Q, Guo Y, et al. Mixcode: Enhancing code classification by mixup-based data augmentation. In: IEEE. 2023:379–390.

43. Gal Y, Ghahramani Z. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In: PMLR. 2016:1050–1059.

44. Houlsby N, Huszár F, Ghahramani Z, Lengyel M. Bayesian active learning for classification and preference learning. *arXiv preprint arXiv:1112.5745.* 2011.

45. Zhu F, Cheng Z, Zhang XY, Liu CL. Rethinking confidence calibration for failure prediction. *arXiv preprint arXiv:2303.02970.* 2023.

46. Kuhn L, Gal Y, Farquhar S. Semantic uncertainty: Linguistic invariances for uncertainty estimation in natural language generation. *arXiv preprint arXiv:2302.09664.* 2023.

47. Huang Y, Song J, Wang Z, Chen H, Ma L. Look before you leap: An exploratory study of uncertainty measurement for large language models. *arXiv preprint arXiv:2307.10236.* 2023.