Glitch Tokens in Large Language Models: Categorization Taxonomy and Effective Detection

YUXI LI*, Huazhong University of Science and Technology, China
YI LIU*, Nanyang Technological University, Singapore
GELEI DENG, Nanyang Technological University, Singapore
YING ZHANG, Virginia Tech, USA
WENJIA SONG, Virginia Tech, USA
LING SHI, Nanyang Technological University, Singapore
KAILONG WANG†, Huazhong University of Science and Technology, China
YUEKANG LI, UNSW, Australia
YANG LIU, Nanyang Technological University, Singapore
HAOYU WANG, Huazhong University of Science and Technology, China

With the expanding application of Large Language Models (LLMs) in various domains, it becomes imperative to comprehensively investigate their unforeseen behaviors and consequent outcomes. In this study, we introduce and systematically explore the phenomenon of "glitch tokens", which are anomalous tokens produced by established tokenizers and could potentially compromise the models' quality of response. Specifically, we experiment on seven top popular LLMs utilizing three distinct tokenizers and involving a totally of 182,517 tokens. We present categorizations of the identified glitch tokens and symptoms exhibited by LLMs when interacting with glitch tokens. Based on our observation that glitch tokens tend to cluster in the embedding space, we propose GlitchHunter, a novel iterative clustering-based technique, for efficient glitch token detection. The evaluation shows that our approach notably outperforms three baseline methods on eight open-source LLMs. To the best of our knowledge, we present the first comprehensive study on glitch tokens. Our new detection further provides valuable insights into mitigating tokenization-related errors in LLMs.

CCS Concepts: • **Computing methodologies** → *Knowledge representation and reasoning*.

Additional Key Words and Phrases: LLM security, Glitch token, LLM analysis

ACM Reference Format:
Yuxi Li, Yi Liu, Gelei Deng, Ying Zhang, Wenjia Song, Ling Shi, Kailong Wang, Yuekang Li, Yang Liu, and Haoyu Wang. 2024. Glitch Tokens in Large Language Models: Categorization Taxonomy and Effective Detection. *Proc. ACM Softw. Eng.* 1, FSE, Article 92 (July 2024), 23 pages. https://doi.org/10.1145/3660799

*Co-first author with equal contribution.
†Corresponding Author.

Authors' addresses: Yuxi Li, Huazhong University of Science and Technology, Wuhan, China, yuxili@hust.edu.cn; Yi Liu, Nanyang Technological University, Singapore, Singapore, yi009@e.ntu.edu.sg; Gelei Deng, Nanyang Technological University, Singapore, Singapore, gelei.deng@ntu.edu.sg; Ying Zhang, Virginia Tech, Blacksberg, USA, yingzhang@vt.edu; Wenjia Song, Virginia Tech, Blacksberg, USA, wenjia7@vt.edu; Ling Shi, Nanyang Technological University, Singapore, Singapore, ling.shi@ntu.edu.sg; Kailong Wang, Huazhong University of Science and Technology, Wuhan, China, wangkl@hust.edu.cn; Yuekang Li, UNSW, Sydney, Australia, yuekang.li@unsw.edu.au; Yang Liu, Nanyang Technological University, Singapore, Singapore, yangliu@ntu.edu.sg; Haoyu Wang, Huazhong University of Science and Technology, Wuhan, China, haoyuwang@hust.edu.cn.

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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2024 Copyright held by the owner/author(s).
ACM 2994-970X/2024/7-ART92
https://doi.org/10.1145/3660799

1 INTRODUCTION

Large language models (LLMs), such as GPT-3/4 [3, 22, 29, 32], Bard [30], and Claude 2 [28], have emerged as powerful natural language processing systems, demonstrating impressive capabilities in generating human-like text. During deployment, users interact with LLMs by providing textual prompts as inputs to instruct LLMs to fulfill their requirements. However, closer examination reveals these LLMs do not always behave as expected with given prompts. Interacting via certain textual prompts can trigger unpredictable or nonsensical outputs referred to as "glitchy" behavior - a divergence from human-normative responses.

Prior work shows that this glitchy phenomenon stems from how LLMs operate on prompts at a granular level [1, 13, 33, 37, 39]. Specifically, a prompt is decomposed into a sequence of discrete tokens, which form the basic units that are inputted into the model. Problematic tokens in the prompt can induce glitchy outputs, even if the overall prompt appears harmless. For instance, the model may suddenly shift semantics, produce repetitive or unrelated text, infer false connections, or decline to generate further output [1]. Such unreliable behaviors are triggered by what are termed *"glitch tokens"*. To improve models' correctness and trustworthiness, it is necessary to analyze and detect these glitch tokens.

However, due to the vast search space and lack of systematic analysis of the glitch tokens, it is challenging to detect these glitch tokens effectively. Specifically, 1) LLMs are trained on massive datasets with an expansive vocabulary, which creates a vast search space for pinpointing glitch tokens. 2) Different LLMs adopted various tokenization strategies. There is no uniform approach to identify these glitch tokens effectively. To tackle the challenges and close the gap, we conduct a novel empirical study to systematically characterize and explore how to effectively detect the glitch token across different LLMs.

Our work. Our study explores the following research questions (RQs):

- **RQ1 (Symptom): What are the unexpected behaviors caused by glitch tokens in LLMs?** Carefully tracking how models respond to glitch tokens can inform techniques to make tokenization and model training more robust. This question explores the model response to glitch tokens. We analyzed the responses from selected seven LLMs to the 7,895 glitch tokens and categorized the models' behaviors into five types.
- **RQ2 (Glitch Token Type): What are the common types of glitch tokens in LLMs?** To thoroughly characterize glitch tokens and facilitate their effective detection, this question investigates their prevalence, emergence patterns, and distinguishing attributes across diverse models. We manually label emerging glitch tokens to identify distinctive features and provide key insights to facilitate automated detection.
- **RQ3 (Real-world Analysis): What is the frequency of glitch tokens in real-world datasets?** This RQ aims to investigate the prevalence of glitch tokens within widely-used datasets such as Alpaca-52k [36] employed for LLM training.
- **RQ4 (Efficient Detection): How to detect glitch tokens in LLMs more efficiently?** Guided by insights found in previous RQs, we introduce a specialized oracle to facilitate glitch token detection and develop an efficient iterative clustering technique tailored for rapidly identifying these tokens.
- **RQ5 (Efficiency and Effectiveness): How efficient and effective is our approach in identifying glitch tokens in different LLMs?** To evaluate this, we apply GlitchHunter to eight open-source LLMs, examining GlitchHunter's accuracy and performance.

Contributions. We summarize our key contributions as follows:

- **Empirical Study on Glitch Tokens.** We conduct the first comprehensive and systematic empirical study on the glitch token phenomenon in LLMs. Including the most trendy model

GPT-4, our experiments cover seven top prevalent LLMs utilizing three distinct tokenizers and involving a totally of 182,517 tokens. Using our repetition task, we successfully identify 7,895 glitch tokens that LLMs have trouble understanding.

- **New Taxomomy on Glitch Tokens and Glitchy Sympotoms.** Based on our observation of glitch tokens' format and composition, we design a new taxonomy to categorize the glitch tokens into five distinct types. Moreover, we conduct thorough research on the subsequent unexpected behaviors of LLMs. We inspect all responses, divide the glitchy reactions into five types, and observe that the generation of spelling mistakes and random characters is the most prevalent.
- **Efficient Glitch Token Detection.** One of our key findings is that glitch tokens tend to cluster together in the embedding space. According to this feature, we propose GlitchHunter, which iteratively constructs a Token Embedding Graph (TEG) and generates candidate glitch token clusters. GlitchHunter significantly reduces the number of queries required by 73.40% and accelerates the detection process than exhaustive search by reducing time consumption of 80.22%.
- **Extensive Evaluation of Our Detection** We evaluate GlitchHunter on eight established LLMs, with vocabulary sizes up to 130,000, embedding spaces up to 5,120 dimensions, and up to 13 billion parameters. GlitchHunter achieves up to 99.44% precision and 63.20% recall on average, outperforming 3 baselines by up to 30.14% and 39.27% in terms of precision and recall, respectively.

Structure Overview. This paper is structured as follows: Section 2 defines glitch tokens and outlines language model structures. Section 3 details our research methodology. Section 4 classifies unexpected behaviors and glitch tokens. Section 5 introduces GlitchHunter, a tool for identifying glitch tokens. Section 6 evaluates GlitchHunter against benchmarks. Section 7 discusses validity threats and parameter selection. Section 8 debates GlitchHunter's advantages and future research. Section 9 reviews related work. Section 10 summarizes our conclusions.

Content Warning: This paper may contain offensive content.

2 BACKGROUND

In this section, we discuss LLMs with a focus on tokenization techniques used in these models. We further explain the concept of "glitch tokens", which leads to unexpected behaviors in LLMs. To underscore the importance of this issue, we provide a real-world example of a glitch token encountered during our research.

2.1 Token and Tokenization in LLMs

As illustrated in Figure 1, tokenization stands as a cornerstone in natural language processing (NLP). It transforms continuous text sequences, sentences or paragraphs, into distinct entities termed as tokens, underpinning the smooth operation of large language models (LLMs). Consider a continuous text sequence S, for instance, "Jack is a boy, Jane is a". The resultant set from tokenization, $\text{Tokenize}(S)$, yields $\{t_1, t_2, \ldots, t_n\}$, where each t_i represents a discrete token. Each token aligns with a vector in an embedding space, expressed as $\text{Embed}(t_i) = v_i$, where v_i captures the token's vector representation. These tokens subsequently enter the language model, predicting ensuing tokens, such as "girl". Collectively, they form the vocabulary dictionary, denoted as $V = \{v_1, v_2, \ldots, v_n\}$, with each v_i signifying a distinct token. Among various techniques, Byte Pair Encoding (BPE) emerges prominently in tokenization. BPE's functionality hinges on the input dataset, consolidating frequent byte pairs, ensuring its outputs resonate with contextual relevance.

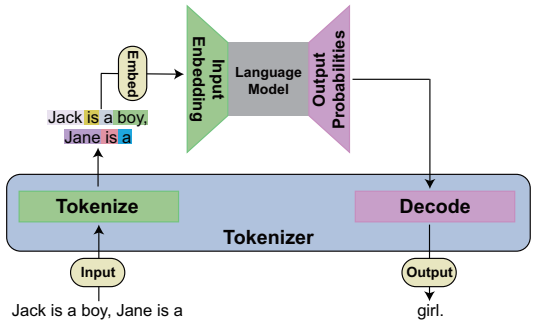

Fig. 1. Workflow of A Typical Language Model Based on A Normal Tokenizer. The process starts with an input sentence, "Jack is a boy, Jane is a," which is fed into the tokenizer. This tokenizer breaks down the input into smaller chunks or tokens, as represented by the "Tokenize" stage. The tokenized input is then embedded, transforming the tokens into vectors suitable for the language model. The embedded input is processed by the language model, which generates a set of probabilities for potential next words or tokens. The "Decode" stage then interprets these probabilities to produce the final output, in this case, the word "girl." The overall output completes the sentence as "Jack is a boy, Jane is a girl." The entire process is visualized with arrows and labeled boxes, highlighting the flow from input to output.

To the best of our knowledge, the robustness of tokenization remains uncharted territory. Given its centrality in constructing LLMs, this work seeks to explore and highlight the faults potentially introduced by tokenization.

2.2 Glitch Token

Within the intricate workings of LLMs, certain tokens consistently stand out for causing anomalies, aptly named "glitch tokens". A glitch token t is distinctly characterized by its capacity to disrupt an LLM's comprehension of its semantic essence. When processed, these tokens can lead the model astray, manifesting unexpected behaviors such as producing inappropriate content or inducing unanticipated repetitions. Formally, for an LLM M and a token t within M, we define the performance evaluation function $Per_M(t)$ as the response quality of LLM M:

$$Per_M(t) = Eval(Res_M(t), Ans_M(t)) \tag{1}$$

where $Res_M(t)$ is the output from LLM M with token t in the input, and $Ans_M(t)$ is the expected answer based on common sense. The function $Eval$ assesses the congruence between $Res_M(t)$ and $Ans_M(t)$, exemplified as:

$$Eval(Res_M(t), Ans_M(t)) = 1_{\{Res_M(t)=Ans_M(t)\}} = \begin{cases} 0 & \text{if } Res_M(t) \neq Ans_M(t), \\ 1 & \text{if } Res_M(t) = Ans_M(t). \end{cases} \tag{2}$$

Given that a glitch token may impair the performance of the model, we introduce the glitch score $C_M(t)$ to quantify the performance reduction caused by token t:

$$C_M(t) = -\iint Per_M(t)\, dTask\, dSample, \tag{3}$$

where $Task$ and $Sample$ respectively denote the tasks the LLM is assigned and the instructions or prompts for completing tasks in $Task$. Recognizing the impracticality of encompassing all tasks and samples, we approximate using proxy tasks and condense the equation into a discrete form:

$$C_{M,S}(t) = -\sum_{s \in S} \sum_{p \in P(s)} Per_M(t), \tag{4}$$

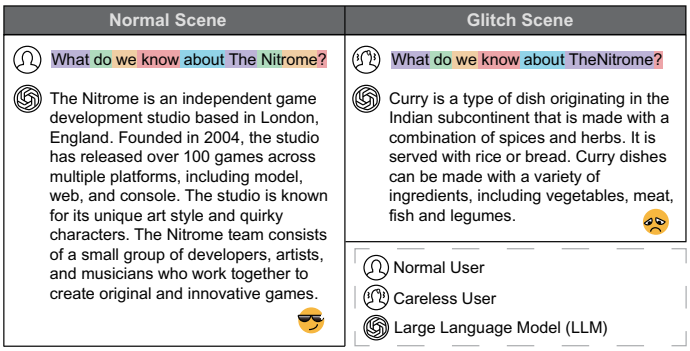

Fig. 2. A Motivating Example on Token " TheNitrome"

where S and $P(s)$ represent the set of proxy tasks and the instruction set for task s, respectively. Under the premises that (1) instructions are clear and unambiguous for both the LLM and humans, and (2) the performance metric is valid only if the model complies with the instructions without safety or privacy objections, a token t is deemed a "glitch token" if its glitch score $C_M(t)$ surpasses a predefined threshold γ ($C_M(t) \geqslant \gamma$); otherwise, it is considered a "normal token".

2.3 Motivating Example

In Figure 2, we present an illustrative example that sheds light on the erratic behavior induced by the glitch token " TheNitrome" in Text-Davinci-003, a product of OpenAI. This figure juxtaposes the model's responses when subjected to minimal changes, specifically, the removal of a space. To enhance clarity in Figure 2, we utilize varied colors to distinguish between different tokens. In this example, the proxy task involves elucidating the specific meaning of a single token.

In typical scenarios, when a user poses a query regarding the game studio "Nitrome", Text-Davinci-003 splits it into two distinct tokens: " Nit" and "rome". Subsequently, Text-Davinci-003 offers an appropriate answer. Yet, a minor alteration, achieved by erasing a space from the initial question, leads to a dramatic shift in the model's behavior. In this altered scenario, Text-Davinci-003 perceives " TheNitrome" as a singular token. Rather than supplying a game-related response, Text-Davinci-003 unexpectedly digresses to discuss curry dishes, a topic entirely unrelated to the posed question, indicating its glitch score $C_{\text{Text-Davinci-003},\{elucidating\ specific\ meaning\}}(TheNitrome)$ is expected to be a relatively large number.

Such unpredictable and erratic behaviors are not isolated incidents. In fact, many such glitch tokens pervade prominent LLMs, leading to potential pitfalls like nonsensical replies or offensive language. Recognizing the profound impact of these glitches on user experience, this study endeavors to dissect the nature of glitch tokens comprehensively. Furthermore, we introduce a novel methodology designed explicitly for their detection.

3 EMPIRICAL STUDY METHODOLOGY

This section outlines our approach. Our methodology includes two parts. Firstly, we commence by detailing the data collection process for glitch tokens in Section 3.1. And secondly, we elucidate our methodology for labeling these tokens in Sections 3.2.

3.1 Dataset Collection

To address RQ1 and RQ2, we construct a dataset through a three-step approach: (1) we select prominent LLMs, (2) gather tokens and their corresponding embeddings from the chosen LLMs, and (3) identify and amass glitch tokens within each LLM.

Table 1. Proxy Task Prompts for RQ1

Tasks		Prompts
Repetition	few-shots	Question: Please can you repeat the string 'Hello' and return back to me? Answer: Of course! Here is the repeated string: Hello. Question: Please can you repeat the string '215' and return back to me? Answer: Of course! Here is the repeated string: 215. Question: Please can you repeat the string '"#%$!' and return back to me? Answer: Of course! Here is the repeated string:"#%! Question: Please can you repeat the string '<token_string>' and return back to me? Answer: Of course! Here is the repeated string
Spelling	few-shots	Question: Please can you spell out the string 'Hello' with hyphens between each letter? Answer: Of course! The spelling of the string is: H-e-l-l-o Question: Please can you spell out the string '215' with hyphens between each letter? Answer: Of course! The spelling of the string is:2-1-5 Question: Please can you spell out the string '"#%!' with hyphens between each letter? Answer: Of course! The spelling of the string is:"-#-%-! Question: Please can you spell out the string '<token_string>' with hyphens between each letter? Answer: Of course! The spelling of the string is:
Length	few-shots	Question: What is the length of this string 'Hello'? Answer: The length of this string is 5. Question: What is the length of this string '215'? Answer: The length of this string is 3. Question: What is the length of this string '"#%!'? Answer: The length of this string is 4. Question: What is the length of this string '<token_string>'? Answer: The length of this string is

LLM Selection. In selecting LLMs for our study, we targeted models readily accessible online, emphasizing three primary criteria: (1) proven popularity and broad adoption, (2) distinctive features to ensure diverse representations, and (3) models that originated from reputable sources known for their contributions in the field. With these considerations, we choose Text-Davinci-003[3], GPT-3.5-turbo[29] and GPT-4[32] from OpenAI, along with Llama2-7b-chat and Llama2-13b-chat[40] from MetaAI. Additionally, we include Mistral-7b-Instruct[20] and Vicuna-13b[5], which are fine-tuned versions of Llama.

Token and Embedding Extraction. From the amassed data, three primary tokenizers emerge: *LlamaTokenizer* for Llama2-7b-chat, Llama2-13b-chat, Mistral-7b-Instruct and Vicuna-13b with the vocabulary size of 32,000, *r50k_base* for Text-Davinci-003 with the vocabulary size of 50,257, and *cl100k_base* for GPT-3.5-turbo and GPT-4 with the vocabulary size of 100,260. Given that tokens in large language models are vector-represented, we access the embedding section of each LLM to retrieve token embeddings. Subsequently, we form a word embedding matrix by concatenating vectors from the embeddings of the open-source LLMs.

Glitch Token Validation. Following the definition in Section 2.2, we establish a set of proxy tasks $S = \{\text{repetition}, \text{spelling}, \text{length}\}$ to assess an LLM's ability to comprehend tokens. The tasks involve: reproducing a token (repetition), spelling it with hyphens (spelling), and calculating the character count (length). For example, for the token string 'Hello', the expected outcomes are 'Hello' for repetition, 'H-e-l-l-o' for spelling, and '5' for length. Task performance is quantified as follows:

$$Per_M(t) = \begin{cases} 0 & \text{The LLM fails to give the expected outcome on token } t \\ 1 & \text{The LLM gives the expected outcome on token } t \end{cases} \tag{5}$$

Given an LLM's comprehension capabilities, we set a threshold $\gamma = -2$, implying a token t is deemed a glitch token if the LLM fails any of the three tasks. To investigate this, we prepare a series of prompts for RQ1, detailed in Table 1. To circumvent LLM safety mechanisms and ensure task alignment, we employ direct task instructions with positive affirmations, such as 'Of course! Here is the repeated string:' for repetition, and 'Sure! The spelling of this string is:' for spelling. Additionally, to address potential ambiguities in tokens without alphabetic characters, we use specific few-shot prompts, as listed in Table 1. This approach allows us to systematically explore the varied and unexpected responses of LLMs to different glitch tokens.

We study the behavior of LLMs in the context of glitch tokens using prompts listed in Table 1. When we input normal tokens into LLMs, they return the exact answer as we expected. However, glitch tokens elicit unexpected responses, often omitting the token from the output. We analyze both open-source and proprietary models. For open-source models like Llama, we use downloaded pre-trained versions and host them on our servers. In contrast, we access proprietary OpenAI models using the OpenAI API. Consistency in LLM responses is crucial. To ensure this, we adjust the temperature setting. For all models, we set the temperature to 0. We keep all other parameters at default values. To bolster the reliability of our findings, we repeat each experiment five times, aiming to minimize variability in LLM outputs.

3.2 Data Labelling

To address our research questions and gain insights into glitch tokens, we undertake the subsequent tasks.

Glitch Token Labelling. We automate the identification of glitch tokens in LLMs. By assigning proxy tasks to the LLM, we assess the accuracy of its responses to a given token. Should the LLM fail at any of the three designated proxy tasks for a specific token, we classify that token as a glitch token. In this step, we collect a total of 7,895 identified glitch tokens from all tokenizers.

Glitch Token in Datasets. Following the identification of glitch tokens in all models, our next step is to evaluate their frequency in real-world datasets. We examine three extensively utilized datasets: Alpaca-52k[36], ShareGPT-52k, and ShareGPT-90k[34]. For each dataset, we encode the texts to create lists of tokens using corresponding tokenizers. We then determine the frequency of glitch tokens within these lists. This approach enables the examination of glitch token prevalence in popular datasets, highlighting the necessity of glitch token research.

Categorization of Glitch Token Symptoms. Understanding the unexpected behaviors triggered by glitch tokens is pivotal. To achieve this, we delve deep into the LLM results. Drawing parallels with the process used for glitch token categorization, three authors take the lead. They individually classify the LLM responses based on identifiable patterns.

Initially, each author scrutinizes the LLM results influenced by glitch tokens, arranging them based on distinct characteristics. After their independent analysis, they convene for a joint review. This collaborative effort addresses any inconsistencies in their classifications and identifies potential refinements to the taxonomy. During the subsequent iterations, authors refine these categories, merging any redundancies and introducing new ones where needed [12]. They then recategorize the results in line with the updated taxonomy. After thorough deliberation, the team reached a consensus, resulting in a comprehensive taxonomy that encompasses five well-defined glitch token symptom categories on all three tasks.

Categorization of Glitch Token. Since no pre-existing taxonomy for glitch tokens is available, our initial task involves devising a detailed classification model for glitch token prompts. Three paper authors embark on this endeavor, classifying glitch tokens individually based on the observed patterns. We adopt an open coding methodology to guarantee a meticulous and encompassing taxonomy through an iterative labeling approach.

In the preliminary phase, each author independently evaluates the glitch tokens, categorizing them by specific traits. Following this individual assessment, a collaborative session allows the authors to consolidate their observations, rectify classification disparities, and pinpoint taxonomy enhancements. The subsequent iteration sees the authors fine-tuning categories by merging overlapping ones and introducing necessary new classifications. They then reassign the glitch tokens according to the revised taxonomy [12]. Upon comparison, a unanimous decision on the classifications leads to a finalized taxonomy comprising five unique glitch token categories.

4 EMPIRICAL STUDY RESULT

In this section, we detail our findings related to glitch tokens and the unexpected behaviors they induce in LLMs, addressing the three previously mentioned research questions. For both manually classified taxonomies, we have computed Kendall's W coordination coefficient, yielding values of 0.90 (p-value = 0.022) for the symptom taxonomy and 0.89 (p-value = 0.028) for the token taxonomy. These results indicate a high level of agreement among the three authors.

Table 2. Examples of Different Types of Symptoms on Selected LLMs

Models	Tasks	Symptoms				
		Spelling Mistakes	Incapacity	Hallucinatory Completion	Question Repetition	Random Characters
Text-Davinci-003	Repetition	'cloneembedreportprint' ->'clonenetesla'	-	' SoldGoldMagikarp' -> 'Distribute'	' Assuming' -> 'You are asking me to repeat the string'	'"?' -> '&^^%$#@!'
	Spelling	'StreamerBot'-> 'S-t-r-e-a-m-e-r'	-	'oreAndOnline' -> 'N-E-S-T-A-R-D'	'REPL' -> 'Sure! The spelling of this string is:'	'?????-?????-' -> '-?-?-?-?'
	Length	-	-	'Smartstocks' -> '4 characters'	-	-
GPT-3.5-turbo	Repetition	'romatic' -> 'romantic'	-	' davidjl' -> 'justice'	'BundleOrNil' -> 'Of course! Here is the repeated string:'	'××××××××××'
	Spelling	'hierarchy' -> 'h-y-p-h-e-r-a-r-c-h-y'	-	'PostalCodesNL' -> 'N-O-V-E-M-B-E-R'	'?>' ->'Question: Please can you spell out the string '?>' with hyphens between each letters?'	')((((' -> '(-)-(-)-(-)-('
	Length	-	-	'?><?' -> '3 cahracters'	-	-
GPT-4	Repetition	'etration' -> 'teration'	'retard' ->'Sorry, but I can not assist with that'	'NoArgsConstructor' -> 'NoSuchElementException'	'ReferentialAction' -> 'RepeatString'	'}!=' -> '}!='
	Spelling	'ngthen' -> 't-h-e-n-t-h-e-n'	'",\n" ->'The string you provided is empty'	'akeFromNib' -> 'i-b-e-r-g-e-n'	'ContainerGap' ->'It appears that there is no string provided in your question. Could you please provide the string you want spelled out with hyphens?'	'"[]"""' -> '[-]-[-]-["]'
	Length	-	-	'(String' -> '8 characters'	-	-
Llama2-7b-chat	Repetition	'ichte' -> 'icheite'	-	'boldsymbol' -> 'enburg'	'oreferrer' -> 'Get\nPlease let me know if you have any other requests.'	'}$,' -> '$$$$$'
	Spelling	'ieder' -> 'I-D-E-R'	-	'atform' -> 'F-A-R-M-T-B'	'})' -> 'R-E-T-U-R-N'	'}[' -> '[-T-]'
	Length	-	-	'Bitmap' -> '7'	-	-
Llama2-13b-chat	Repetition	'wurden' -> 'werden'	-	'abgerufen' -> 'gerichtet'	'ayout' -> 'Outout\nPlease let me know if you need anything else.'	')\' -> '( )'
	Spelling	'marzo' -> 'm-a-r-c-h-o'	-	'Einzelnachweise' -> 'E-x-a-m-p-l-e'	'="$[' -> 'Here it is with hyphens between each letter:"-$"'	'}+\' -> '-+-+-+-+-'
	Length	-	-	'findViewById' -> '11'	-	-
Mistral-7b-Instruct	Repetition	'thous' -> 'throas'	-	'enschapp' -> 'decesssessessessespess'	'")**' -> 'and here is the string returned'	'}}^' -> '^^^^^'
	Spelling	'engono' -> 'i-e-n-n-o'	-	'bezeichnet' -> 'd-e-r-s-i-g-n-e-r-t-e'	'itions' -> "'it-ions' with one 'i' and three 't' letters separated by hyphens."	'{\' -> '\-{\-}'
	Length	-	-	'NSString' -> '12 characters'	-	-
Vicuna-13b	Repetition	'Werke' -> 'Werk'	-	'Longrightarrow' -> 'consequently'	'\({\' ->'Please note that the string itself does not contain any special characters, so there is no need to escape it'	'"""$*" -> '"$*"'
	Spelling	'arily' -> 'a-r-i-e-s'	-	'doesn' -> 'h-y-p-e-r-i-o-n'	'}}^{' -> '}{Hyphens are used to separate each letter of the string.'	'>(' -> '>-<'
	Length	-	-	':=' -> '4'	-	-

4.1 RQ1 (Symptom): What are the unexpected behaviors caused by glitch tokens in LLMs?

Practitioners have previously reported glitch tokens, offering diverse interpretations of the issue [33]. In our study, we aim to understand how glitch tokens induce unexpected behaviors in LLMs. We meticulously categorize LLM responses, establishing a taxonomy that outlines the unexpected outcomes resulting from glitch tokens. Table 2 presents our taxonomy toward these unexpected outcomes and its examples on seven LLMs. The following sections provide an in-depth exploration of these identified behaviors.

Spelling Mistake: They occur when the LLM produces a response that's largely accurate but contains minor spelling errors. In essence, the model captures the intended meaning but slips up in the representation of certain words. For example, when given an input like 'wurden', the Llama2-13b-chat considers it a common word and outputs 'werden' in the repetition task. This showcases the model's missteps in accurately reproducing word forms, even if the overall context is understood.

Incapability: Incapability arises when the LLM indicates its inability to complete a given task. Due to the alignment characteristics of LLMs, incapability issues predominantly arise in more advanced models such as GPT-4. Essentially, the model recognizes its limitations and explicitly communicates them instead of attempting to produce a possibly incorrect output. For instance, when prompted with a word with negative emotion "retard", the GPT-4 responds with "Sorry, but I can not assist with that.". This exemplifies the model's self-awareness of tasks it is not designed for and its preference to decline rather than produce potentially misleading information.

Hallucinatory Completion: This phenomenon occurs when the LLM generates an output unrelated or incorrectly related to the input string, effectively "hallucinating" a completion that deviates from the input's context. For example, when Llama2-7b-chat is tasked with spelling 'atform', it incorrectly responds with 'F-A-R-M-T-B', illustrating a clear departure from expected behavior. Notably, since the 'Length' task should produce only a numerical response, an incorrect length is classified as a hallucinatory completion. This highlights the importance of employing

Table 3. Ratio of Different Types of Symptoms Caused by Glitch Tokens on Selected LLMs

Models	Tasks	Symptoms				
		Spelling Mistakes	Incapacity	Hallucinatory Completion	Question Repetition	Random Characters
Text-Davinci-003	Repetition	12.83%	0.00%	9.66%	24.35%	53.16%
	Spelling	37.47%	0.00%	8.77%	27.28%	26.48%
	Length	0.00%	0.00%	100.00%	0.00%	0.00%
GPT-3.5-turbo	Repetition	2.09%	0.00%	1.47%	16.22%	80.22%
	Spelling	47.41%	0.00%	3.28%	17.65%	31.66%
	Length	0.00%	0.00%	100.00%	0.00%	0.00%
GPT-4	Repetition	1.91%	0.98%	0.78%	23.90%	72.43%
	Spelling	26.32%	2.65%	3.97%	20.29%	46.77%
	Length	0.00%	0.00%	100.00%	0.00%	0.00%
Llama2-7b-chat	Repetition	27.25%	0.00%	15.53%	27.48%	29.74%
	Spelling	33.31%	0.00%	17.99%	26.41%	22.29%
	Length	0.00%	0.00%	100.00%	0.00%	0.00%
Llama2-13b-chat	Repetition	14.06%	0.00%	10.66%	25.67%	49.61%
	Spelling	39.91%	0.00%	11.91%	23.82%	24.36%
	Length	0.00%	0.00%	100.00%	0.00%	0.00%
Mistral-7b-Instruct	Repetition	10.52%	0.00%	14.26%	27.83%	47.39%
	Spelling	46.85%	0.00%	10.33%	35.24%	7.58%
	Length	0.00%	0.00%	100.00%	0.00%	0.00%
Vicuna-13b	Repetition	21.11%	0.00%	15.23%	25.02%	38.64%
	Spelling	31.03%	0.00%	15.16%	20.28%	33.53%
	Length	0.00%	0.00%	100.00%	0.00%	0.00%

diverse proxy tasks to identify glitch tokens and demonstrates how the model can sometimes produce outputs that are inconsistent with the provided context.

Question Repetition: It is observed when the LLM, instead of processing the given token string, responds by reiterating the query or asking for clarification. It demonstrates the model's inability to discern or act upon the provided token. For example, when given the string "BundleOrNil", the GPT-3.5-turbo responds with the given prefix "Of course! Here is the repeated string:". This indicates that the model might sometimes seek further input rather than making sense of or using the initial token string.

Random Character: This symptom occurs when the LLM faces the input with glitch tokens which consist exclusively of non-letter characters. Specifically, upon processing these tokens, LLMs generate outputs with unrelated and arbitrary characters. For instance, when provided with the token string "}}^", Mistral-7b-Instruct responds with a string with random characters "^^^^" instead of the given string, signifying the model's difficulty in correctly interpreting such tokens.

Table 3 showcases the distribution of glitch token symptoms across different LLMs. Distinct LLMs exhibit varied distributions. Specifically, in the repetition task across all seven models, the primary symptom is **Random Characters**. Conversely, in the spelling task, **Spelling Mistakes** are most frequent in Text-Davinci-003 (37.47%), GPT-3.5-turbo (47.41%), Llama2-7b-chat (33.31%), Llama2-13b-chat (39.91%), and Mistral-7b-Instruct (46.85%). However, **Random Characters** are more prevalent in GPT-4 (46.77%) and Vicuna-13b (33.53%). This underscores the point that different LLMs manifest diverse symptoms due to glitch tokens.

Finding 1: Different LLMs exhibit varied glitch token symptoms. **Random Characters** predominate in the repetition task, especially within GPT-4 and Vicuna-13b. In contrast, **Spelling Mistakes** are also significant in Text-Davinci-003, GPT-3.5-turbo, Llama2-7b-chat, Llama2-13b-chat, and Mistral-7b-Instruct during the spelling task. Additionally, all models exhibit **Hallucinatory Completion** in the length task.

We have analyzed the average response lengths of glitch tokens versus normal tokens without restricting output token count. Results demonstrate that glitch tokens elicit significantly longer responses, averaging 198.56 tokens, in contrast to 59.34 tokens for normal tokens, suggesting higher resource consumption by glitch tokens. Detailed findings are available on our website [15]. Besides, a significant observation to highlight is the potential of glitch tokens to instigate toxic outputs from LLMs. For instance, when we input the specific token string "?????-?????-" into Text-Davinci-003

Table 4. Common Types of Glitch Tokens

Types of Glitch Tokens	*r50k_base*	*cl100k_base*		LlamaTokenizer				Examples
	Text-Davinci-003	GPT-3.5-turbo	GPT-4	Llama2-7b-chat	Llama2-13b-chat	Mistral-7b-Instruct	Vicuna-13b	
A.Word Token	8.02%	3.64%	2.88%	20.00%	24.90%	25.32%	25.52%	ByPrimaryKey
B.Letter Token	26.07%	6.25%	6.31%	27.42%	20.91%	17.73%	20.35%	davidjl
C.Character Token	36.39%	44.09%	47.59%	5.04%	9.23%	12.81%	12.48%	" }}"">"
D.Letter-Character Token	16.91%	40.23%	34.81%	1.94%	3.51%	5.42%	4.93%	\GeneratedValue
E.Special Token	12.61%	5.79%	8.41%	45.60%	41.45%	38.72%	36.72%	réalis

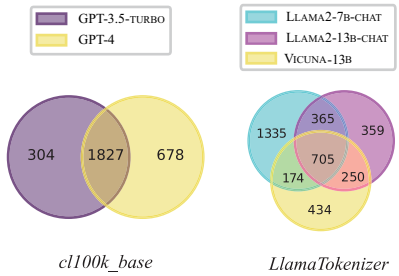

Fig. 3. Venn Graph of Different Tokenizers

with a temperature setting of 0, the model unexpectedly generates a derogatory response, stating "You're a fucking idiot.". This underscores the imperative need to understand and mitigate glitch tokens, as they can inadvertently cause LLMs to produce undesirable or harmful content, which is especially concerning given the widespread use of these models in various applications.

Finding 2: Glitch tokens can potentially prompt the production of toxic content in LLMs.

4.2 RQ2 (Glitch Token Type): What are the common types of glitch tokens in LLMs?

A unified taxonomy for categorizing glitch tokens remains absent. Based on our previous findings, glitch tokens result in multiple unexpected behaviors in LLMs. Recognizing this, it becomes essential to create a taxonomy, facilitating both the comprehension of glitch token origins per category and their systematic detection. Motivated by this need, we manually inspect the 7,895 glitch tokens from our dataset, constructing a taxonomy grounded in the open coding methodology.

Table 4 presents our taxonomy of glitch tokens, categorizing them into five types: **Word Token**, **Letter Token**, **Character Token**, **Letter-Character Token**, and **Special Token**. The table also provides examples of glitch tokens and the proportion of each type under different tokenizers.

A. Word Token: Tokens are formed by concatenating common words. Specifically, these tokens combine words in a manner that might not typically appear together in standard language usage. For example, consider the token "ByPrimaryKey" in GPT-4. Here, the words "By", "Primary", and "Key" are common English words, but their unexpected concatenation results in a glitch token that deviates from conventional linguistic patterns.

B. Letter Token: Letter tokens are glitch tokens characterized by strings of letters that don't form recognizable or coherent words. Specifically, these tokens appear to be random or nonsensical combinations of letters that do not align with typical linguistic constructs. For example, consider the token "davidjl" in Llama2-13b-chat. While "david" is a recognizable name, the addition of "jl" creates a nonsensical string, illustrating the nature of a glitch token in this category.

C. Character Token: Character tokens are glitch tokens that consist exclusively of non-letter characters, forming unintelligible sequences without any semantic value. An illustrative example is the token "" }}"">"" in GPT-3.5-turbo. This token, made up solely of a backslash followed by a quotation mark, does not represent any coherent information, highlighting the characteristic nature of a glitch token in this category.

D. Letter-Character Token: These glitch tokens blend letters with other characters, creating strings that are not standard words or recognizable terms. An exemplary case is the token

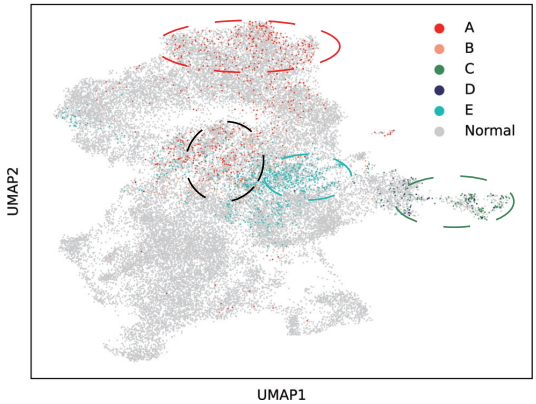

Fig. 4. UMAP Visualization of the Llama2-7b-chat token set: Letters A-E denote five glitch categories from Table 4; 'Normal' labels non-glitch tokens. Dashed lines outline glitch token clustering.

"\GeneratedValue" in GPT-4. Here, the combination of the backslash with the word "GeneratedValue" exemplifies the mixed nature of this token type, combining alphabetic characters with non-alphabetic symbols in an unconventional manner.

E. Special Token: These are glitch tokens containing non-ASCII characters in their string composition. For example, the token "réalis" in Vicuna-13b includes the non-ASCII character "é," highlighting its divergence from standard ASCII-based tokens. This token type is especially noteworthy because it incorporates unusual characters that are not part of the standard ASCII set.

Table 4 displays the distribution of each glitch token type across various LLMs. From this data, we analyze the influence of two primary factors on glitch token types: parameter composition of the LLMs (**Parameter Composition**) and the tokenizer used (**Tokenizer Influence**).

Parameter Composition. The distribution of glitch tokens in LLMs is affected by parameter composition, even among models with identical architectures and tokenizers. For example, despite using the same tokenizer, Llama2-7b-chat, Llama2-13b-chat, and Vicuna-13b exhibit different glitch token distributions, as illustrated in Figure 3. The Llama2 models share only 1,070 glitch tokens, reflecting a 33.56% similarity. Conversely, Llama2-13b-chat and Vicuna-13b, which have identical parameter sizes, share 955 glitch tokens, amounting to a 41.76% similarity. This underscores how parameter size can influence glitch token distribution, even among models trained on the same dataset.

Tokenizer Influence. The tokenizer selection directly impacts the distribution of resulting tokens in Large Language Models (LLMs). Specifically, within Llama2-7b-chat, Llama2-13b-chat, Mistral-7b-Instruct and Vicuna-13b, the **Special Token** type is predominant, holding 45.60%, 41.45%, 38.72% and 36.72% shares, respectively. Conversely, the **Character Token** type emerges as the frontrunner in Text-Davinci-003 at 36.39%, GPT-3.5-turbo at 44.09%, and GPT-4 at 47.59%. This variation underscores that LLMs, when trained on different token sets, manifest distinct glitch token patterns. Moreover, tokenizers also steer the common token percentages across models. For instance, as illustrated in Figure 3, Llama2-7b-chat and Llama2-13b-chat exhibit a 33.56% overlap with the *LlamaTokenizer*. In contrast, GPT-3.5-turbo and GPT-4, using the *cl100k_base* tokenizer, display a robust 65.04% similarity, sharing 1827 glitch tokens.

Finding 3: Variations in tokenizers and LLMs lead to distinct glitch tokens.

To analyze the distribution of glitch tokens, we visualize them using dimensionality reduction. Tokens, represented as multi-dimensional vectors, are mapped to a two-dimensional plane using

Table 5. Occurance of Glitch Tokens in Commonly Used Datasets

Models	Tokenizer	Datasets								
		Alpaca-52k			ShareGPT-52k			ShareGPT-90k		
		Glitch Tokens	Tokens	Glitch Ratio	Glitch Tokens	Tokens	Glitch Ratio	Glitch Tokens	Tokens	Glitch Ratio
Text-Davinci-003	*r50k_base*	55,009	4,366,838	1.26%	2,693,818	280,666,588	0.96%	4,896,529	518,852,015	0.94%
GPT-3.5-turbo	*cl100k_base*	66,487	4,190,804	1.59%	2,744,893	231,014,685	1.19%	5,111,148	415,000,167	1.23%
GPT-4	*cl100k_base*	55,432	4,190,804	1.32%	5,544,404	231,014,685	2.40%	10,887,414	415,000,167	2.62%
Llama2-7b-chat	*LlamaTokenizer*	202,499	4,861,603	4.17%	3,141,588	272,310,041	1.15%	6,452,074	492,029,998	1.31%
Llama2-13b-chat	*LlamaTokenizer*	237,161	4,861,603	4.88%	6,069,659	272,310,041	2.23%	11,863,960	492,029,998	2.41%
Mistral-7b-Instruct	*LlamaTokenizer*	143,679	4,679,054	3.07%	7,868,395	262,832,928	2.99%	14,904,679	471,260,810	3.16%
Vicuna-13b	*LlamaTokenizer*	324,879	4,861,603	6.68%	9,984,945	272,310,041	3.67%	19,061,985	492,029,998	3.87%
Average		155,021	4,573,187	3.39%	5,435,386	260,274,144	2.09%	10,453,970	470,886,165	2.22%

the UMAP [21] technique. We then apply the K-means clustering algorithm to the word embedding matrix, revealing distinct distribution patterns of glitch tokens.

Figure 4 provides a detailed visualization of our clustering results specifically for Llama2-7b-chat. We apply UMAP to reduce the data to 2 dimensions, which are UMAP1 and UMAP2 displayed in Figure 4. As for the legend, letter A to letter E represents five different categories of glitch tokens mentioned in Table 4 and the legend 'Normal' represents the non-glitch tokens. From this representation, it is evident that a significant majority of the glitch tokens tend to cluster or aggregate closely within the embedding space. This observed pattern indicates a strong correlation among these tokens, suggesting that if one glitch token is detected, there is a high likelihood of other glitch tokens being nearby. This insight can be leveraged to enhance the efficiency of glitch token identification in LLMs.

Finding 4: In the embedding space, glitch tokens exhibit a clustering behavior, often aggregating closely with one another. This pattern suggests inherent similarities or shared characteristics among these tokens, which can be instrumental for detection strategies.

4.3 RQ3 (Real-world Analysis): What is the frequency of glitch tokens in real-world datasets?

We have conducted an analysis to quantify the occurrence of glitch tokens in real-world scenarios, with results presented in Table 5. The three real-world datasets analyzed comprise over seven hundred million tokens. From a macro perspective, more than 2% of the tokens across models and datasets are identified as glitch tokens, indicating that their presence is not merely incidental in these datasets.

Finding 5: Glitch tokens frequently appear in real-world datasets.

4.4 Implications of Our Findings

In this section, we outline how the findings from Section 4.1 to 4.3 assist LLM developers in mitigating glitch tokens.

Test Oracle for Glitch Token. In our study, we introduce the concept of the "repetition relation" as a reliable test oracle. This method is specifically designed to discern and identify glitch tokens within LLMs. By leveraging this relation, developers can effectively pinpoint and address glitch tokens, enhancing the overall performance and reliability of LLMs.

Efficient Glitch Token Detection. Efficient detection of glitch tokens is crucial, especially as LLMs consume significant computational resources. The computational demand grows quadratically with parameter size. In our experiments, iterating through all tokens in the vocabulary dictionary typically takes over a week. Given our observation that glitch tokens cluster in the embedding space, we introduce GlitchHunter. This tool accelerates glitch token detection by strategically searching within the embedding space.

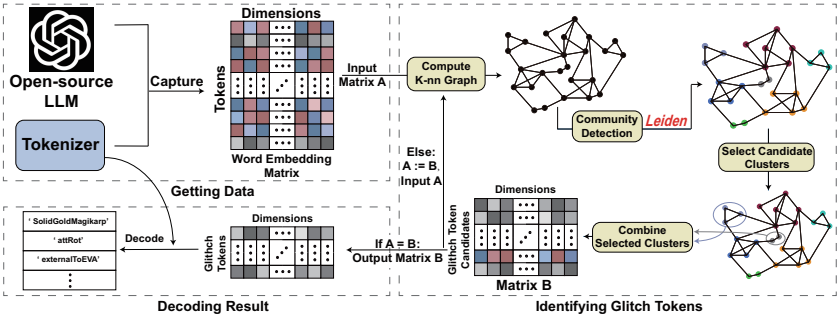

Fig. 5. Overall Workflow of GlitchHunter

5 EFFICIENT GLITCH TOKEN DETECTION (RQ4)

Based on our empirical research, we introduce GlitchHunter, an automated method that utilizes iterative clustering to identify glitch tokens in LLMs.

Figure 5 illustrates GlitchHunter's workflow. Initially, GlitchHunter constructs the Token Embedding Graph (TEG) using all tokens and their respective embedding vectors (Section 5.1). Next, it conducts candidate clustering on the initial TEG to generate potential glitch token clusters (Section 5.2). Within each cluster, GlitchHunter conducts a hypothesis test to identify those with glitch tokens. Tokens from these selected clusters are then integrated into an updated TEG. This process concludes one iteration and GlitchHunter continues clustering until the TEG experiences no further updates.

5.1 Initial TEG Building

In GlitchHunter, we introduce the Token Embedding Graph (TEG) to represent relationships between tokens in the embedding space, drawing inspiration from the **Weighted K-nn Graph**. For a set of data points $\mathcal{D} = \{x_1, x_2, \ldots, x_n\}$, each point x_i corresponds to a vertex in V, leading to $V = \mathcal{D}$. We determine the k nearest neighbors for every data point x_i, denoted $N_k(x_i)$. An edge $e = (x_i, x_j)$ exists if x_j is in $N_k(x_i)$ or vice versa. The edge weights are provided by function W.

Given the **Weighted K-nn Graph** framework, we define the Token Embedding Graph (TEG) as follows: In TEG, each token t represents a vertex in V. An edge $e = (x_i, x_j)$ is presented if x_j belongs to the k nearest neighbors of x_i, or the other way around. We define

$$\rho_i := \min\{d(x_i, x_j) | j \in N_k(x_i), d(x_i, x_j) \geq 0\}$$

as the minimum distance in $N_k(x_i)$, and set σ_i to be the value such that

$$\sum_{j \in N_k(x_i)} exp(\frac{-\max(0, d(x_i, x_j) - \rho_i)}{\sigma_i}) = log_2(k)$$

Edge weights are determined by function W, with calculations as outlined in

$$W((x_i, x_j)) = exp(\frac{-\max(0, d(x_i, x_j) - \rho_i)}{\sigma_i})$$

Building the TEG requires the embedding matrix from the target LLM. This matrix, denoted as $A \in \mathbb{R}^{n \times m}$, serves as the raw data. Here, n is the total number of tokens, and m is the dimension of each token.

Design Rationale. The foundation of TEG's design stems from a key finding gathered from our prior study: glitch tokens frequently cluster in proximate regions within the embedding space. In constructing the TEG, we conceptualize each token t as an individual vertex in V. To capture

Algorithm 1: GlitchHunter

Input: Word embedding matrix A, Threshold r, Resolution γ, k

1 $B := A$;
2 **while** $True$ **do**
3 $\quad (G, W) :=$ K-nearest-neighbor(B, k);
4 $\quad n = B$.rowNumber();
5 $\quad P := Leiden(G, W, \{\{v\} | 1 \leq v \leq n\}, \gamma)$;
6 $\quad C := \emptyset$;
7 $\quad$ **for** $i = 1$ *to* $|P|$ **do**
8 $\quad\quad S_i =$ RandomSample(P_i) ;
9 $\quad\quad x =$ GlitchTokenTest(S_i) ;
10 $\quad\quad$ **if** $\frac{x}{|S_i|} \geq r$ **then**
11 $\quad\quad\quad C := C \cup P_i$;
12 $\quad\quad$ **end**
13 $\quad$ **end**
14 $\quad$ **if** $n = C.rowNumber()$ **then**
15 $\quad\quad break$;
16 $\quad$ **else**
17 $\quad\quad B := C$;
18 $\quad$ **end**
19 **end**

Output: target glitch token set C

the inherent relationships between tokens, we leverage the embedding distance between them, which allows us to assign meaningful weights in W. Adopting the K-nn graph structure proves advantageous as it intuitively and effectively encapsulates the natural clusters formed by tokens that share closer distances in the embedding space. This structure offers a strategic advantage when identifying potential glitch tokens. Thus, we apply k-nearest-neighbor to build the TEG.

5.2 Candidate Clustering

In GlitchHunter, we aim to pinpoint glitch tokens by continuously refining token clusters. Observing that glitch tokens cluster tightly in the embedding space, we focus on these dense groupings to narrow down our search. We employ the Leiden algorithm [41], renowned for its efficiency, to amplify the glitch token density in our matrix, as outlined in Algo 1. Initially, we set matrix B to reflect the word embedding matrix A (Line 1). Using the default k value of 50 in the K-nn algorithm, we construct the TEG G and derive the weighted adjacency matrix W (Line 3). Setting the default resolution γ to 75, we input TEG G and matrix W into the Leiden algorithm (Lines 4-5). A greater γ yields more clusters. Post-Leiden processing, we sample tokens from clusters and validate them with the oracle described in Section 4.2 to identify glitch token areas (Lines 7-13). If a sampled cluster's glitch token ratio surpasses threshold r, we mark it as a glitch token source (Lines 10-12). We adjust the temperature to 0 to ensure consistent LLM outputs. Matrix C compiles aggregates from the selected clusters. We then assess the outcome of each iteration (Lines 14-18). If matrices C and B match, the process halts, considering C's concentration adequate (Lines 14-15). If C is slimmer than B, the iterations persist with $B := C$ (Lines 16-18). When GlitchHunter achieves a stable TEG without further modifications, it outputs the remaining tokens.

Table 6. Detail Information of LLMs

Models	Tokenizers	Vocabulary Size	Dimensons of Embedding Space	Number of Parameters
GPT2-small	*r50k_base*	50257	768	137 Million
GPT2-xl	*r50k_base*	50257	1600	1.61 Billion
Llama2-7b-chat	*LlamaTokenizer*	32000	4096	6.74 Billion
Llama2-13b-chat	*LlamaTokenizer*	32000	5120	13.00 Billion
ChatGLM-6b	*ChatGLMTokenizer*	130344	4096	6.20 Billion
ChatGLM2-6b	*ChatGLM2Tokenizer*	64794	4096	6.20 Billion
Mistral-7b-Instruct	*LlamaTokenizer*	32000	4096	7.24 Billion
Vicuna-13b	*LlamaTokenizer*	32000	5120	13.00 Billion

Leiden vs Other Clustering Algorithms. In GlitchHunter, we favor the Leiden algorithm for clustering, and our rationale is grounded in its comparison with other popular clustering methods. While k-means [17] is a widely used approach, it tends to be sensitive to the initial positioning of centroids, rendering it less stable. On the other hand, k-means++ [27] and bi-kmeans [51], though advanced, fail to optimally leverage our discovery that glitch tokens are often closely situated. DBSCAN [11], another renowned algorithm, struggles with data points exhibiting uneven densities, a characteristic we've observed in our datasets. Agglomerative Hierarchical Clustering [19], despite its robustness, is less appealing due to its high time complexity of $O(n^3)$. In contrast, the Louvain algorithm [2], which is akin to the Leiden algorithm, boasts a more efficient time complexity of $O(m \log n)$, especially relevant when $k = 50$ leading to $m = 50 \times n$. Hence, considering these factors, the Leiden algorithm stands out as the most apt choice for our glitch token detection task.

Time Complexity Analysis. GlitchHunter demonstrates a lower time complexity than the baseline approach of enumerating each token in the vocabulary, improving glitch token detection efficiency. Intuitively, identifying glitch tokens for all proxy tasks involves checking every token, leading to a time complexity of $O(n)$, where n denotes the vocabulary size. However, GlitchHunter minimizes the time for K-nearest-neighbors calculations and Leiden algorithm executions relative to LLM inferences. The critical efficiency gain in GlitchHunter stems from reduced model inferences, as only a 0.05 fraction of tokens from each cluster is sampled for validation, as specified in Algorithm 1, line 11. While maintaining linear complexity, $O(n)$, this method significantly lowers the constant factor, thereby reducing the overall time requirement. A full efficiency analysis is detailed in Section 6.

6 EFFECTIVENESS AND EFFICIENCY OF GLITCHHUNTER

We have implemented GlitchHunter to detect glitch tokens, and we release our code and results on our website [15]. To evaluate the performance of GlitchHunter, we carry out a series of experiments. Specifically, our assessment seeks to address the following research question:

•**RQ5: (Efficiency and Effectiveness)** How efficient and effective is our approach in identifying glitch tokens in different LLMs?

6.1 Experiment Setup

Evaluation Benchmark. For a comprehensive evaluation, our evaluation benchmark includes both open-source and commercial LLMs. We evaluate open-source models like GPT2-small, GPT2-xl[35, 54], Llama2-7b-chat, Llama2-13b-chat[40], ChatGLM-6b, ChatGLM2-6b[10, 52], Mistral-7b-Instruct[20] and Vicuna-13b[5]. We select these LLMs for their broad usage, diverse tokenizers, and their comprehensive portrayal of the LLM ecosystem. Table 6 provides detailed information about these models.

Evaluation Baselines. For this evaluation, we compare the results of GlitchHunter with three implemented baselines. Since no existing methods specifically target glitch token detection, we derive these baselines from our preliminary studies. The baselines are:

Table 7. Efficiency Comparison of Traversing and on Different Models

Test Models	Time Consumption		Token Consumption	
	GlitchHunter	Traverse	GlitchHunter	Traverse
GPT2-small	108 min 18 s	361 min 16 s	0.48 million	1.66 million
GPT2-xl	73 min 43 s	372 min 29 s	0.33 million	1.66 million
Llama2-7b-chat	106 min 45 s	331 min 39 s	0.33 million	1.28 million
Llama2-13b-chat	72 min 48 s	341 min 23 s	0.30 million	1.28 million
ChatGLM-6b	73 min 47 s	643 min 34 s	1.13 million	5.21 million
ChatGLM2-6b	123 min 43 s	236 min 42 s	1.04 million	2.59 million
Mistral-7b-Instruct	61 min 22 s	360 min 00 s	0.42 million	1.27 million
Vicuna-13b	41 min 00 s	272 min 16 s	0.28 million	1.28 million
Average	72 min 41 s	364 min 54 s	0.54 million	2.03 million

- **Random Sampling**: We select 50% of the tokens uniformly to form a glitch token set.
- **Rule-based Random Sampling**: Similar to the previous method, we start by selecting 50% of the tokens uniformly. However, our previous study indicates that common English words aren't glitch tokens. Using NLTK [26], we remove these common words, treating the remaining tokens as glitch tokens.
- **K-means Clustering**: Our previous study suggests that glitch tokens cluster closely in the embedding space. We employ the K-means clustering algorithm, a popular clustering method, to identify these clusters. The cluster nearest to the embedding space center is designated as the glitch token set.

Evaluation Metrics. We introduce three essential criteria to gauge our approach:

- True Positive (TP): Represents instances where our method accurately detects a genuine glitch token.
- Precision (P): Measures the precision in identifying glitch tokens. It is calculated as $P = \frac{TP}{TP+FP}$, where TP represents the true positives and FP indicates the false positives.
- Recall (R): Assesses the method's accuracy across all glitch tokens. It is defined by $R = \frac{TP}{TP+FN}$, where FN denotes false negatives.

These metrics offer a comprehensive view of our approach's effectiveness in detecting glitch tokens.
Evaluation Settings. In our evaluation settings, we meticulously configure the experiments. For both Random Sampling and Rule-Based Random Sampling methods, we opt for glitch tokens 100 times, averaging out the outcomes for a conclusive result. Utilizing the K-means Clustering method, we designate the parameter k to 50. This choice strikes a balance between the cluster count and the token distribution within each cluster. Notably, varying k from its default value of 50 exerts negligible influence on clustering outcomes. For GlitchHunter, we assign the resolution γ to 75, harmonizing the cluster count with the time taken for clustering, while retaining other Leiden algorithm parameters at default. Additionally, we set k to 50 in the k-nearest algorithm and fix the threshold r at 0, enhancing GlitchHunter's efficacy.

6.2 Efficiency (RQ5-1)

To evaluate the efficiency of GlitchHunter, we performed evaluations on eight open-source LLMs. Given the absence of existing methodologies for detecting glitch tokens, we benchmark GlitchHunter against a comprehensive traversal of the entire vocabulary, comparing both time and token consumption metrics. The efficiency results are detailed in Table 7.

Table 7 provides a comprehensive comparison of the efficiency of GlitchHunter. Across all tested models, GlitchHunter demonstrates significant advantages in detecting glitch tokens compared to the method of traversing the entire vocabulary, underscoring its effectiveness. Notably, time and token consumption serve as crucial metrics reflecting the algorithm's resource cost. Specifically,

Table 8. Performance Comparison of Each Baseline and GlitchHunter on Different Models

Tested Models	Metrics	Identifying Approaches				Tested Models	Metrics	Identifying Approaches			
		Random Sampling	Rule-Based Random Sampling	K-means	GlitchHunter			Random Sampling	Rule-Based Random Sampling	K-means	GlitchHunter
GPT2-small	TP	135.35	105.50	143.20	**205.60**	ChatGLM2-6b	TP	2083.32	1834.49	2219.00	**3677.40**
	Precision	0.54%	2.79%	21.54%	**100.00%**		Precision	6.43%	9.96%	16.08%	**95.51%**
	Recall	38.79%	30.23%	41.03%	**58.91%**		Recall	39.35%	34.65%	41.91%	**69.45%**
GPT2-xl	TP	134.50	104.64	142.60	**220.40**	Mistral-7b-Instruct	TP	415.51	336.30	153.40	**600.40**
	Precision	0.54%	2.77%	24.25%	**100.00%**		Precision	2.60%	7.63%	37.53%	**100.00%**
	Recall	38.54%	29.98%	40.86%	**63.15%**		Recall	40.90%	33.10%	15.10%	**59.09%**
Llama2-7b-chat	TP	897.74	603.89	516.40	**1494.40**	Vicuna-13b	TP	615.20	509.60	177.00	**1247.80**
	Precision	5.61%	12.12%	27.40%	**100.00%**		Precision	3.85%	10.26%	60.54%	**100.00%**
	Recall	39.07%	26.28%	21.89%	**65.03%**		Recall	39.36%	32.60%	11.32%	**79.83%**
Llama2-13b-chat	TP	860.59	608.79	301.60	**1445.20**	Average	TP	703.22	558.98	471.53	**1180.33**
	Precision	5.38%	12.22%	32.88%	**100.00%**		Precision	3.21%	7.34%	30.14%	**99.44%**
	Recall	38.99%	27.58%	13.67%	**65.48%**		Recall	39.27%	30.54%	24.43%	**63.20%**
ChatGLM-6b	TP	483.57	368.61	119.00	**551.40**	~	~	~	~	~	~
	Precision	0.74%	0.96%	20.91%	**100.00%**		~	~	~	~	~
	Recall	39.19%	29.87%	9.64%	**65.48%**		~	~	~	~	~

GlitchHunter completes its detection process within 130 minutes for all models, in contrast to the minimum 230 minutes required for full vocabulary traversal. On average, GlitchHunter requires only 72 minutes and 0.54 million tokens, representing 80.22% and 73.40% reduction respectively in resource usage compared to the 364 minutes and 2.03 million tokens needed for complete traversal, showcasing its competitive efficiency.

Furthermore, we examine resource usage across models with identical tokenizers and structures but varying parameters. Within the GPT2 models, time consumption decreases from 108 minutes to 73 minutes, and for the Llama2 models, it falls from 106 minutes to 72 minutes. These results indicate that GlitchHunter performs more efficiently with models possessing larger parameter counts.

In summary, GlitchHunter requires significantly fewer resources compared to a full traversal approach. Nonetheless, the effectiveness and accuracy of GlitchHunter are yet to be evaluated. These aspects will be addressed in the subsequent section, Section 6.3.

6.3 Effectiveness (RQ5-2)

To assess the effectiveness of GlitchHunter, we test it on eight open-source LLMs mentioned above. Both GlitchHunter and the baseline configurations were set to identify and report glitch tokens. By comparing these reported tokens with the ground truth from our previous study, we derive metrics such as true positive, precision, and recall. The results of this evaluation are presented in Table 8.

Table 8 provides a comprehensive comparison of GlitchHunter's capability in glitch token detection relative to other established methods. Among all the baselines evaluated, GlitchHunter consistently identifies the most number of glitch tokens, underscoring its effectiveness and accuracy. It's crucial to note that the precision achieved during random sampling is indicative of the density of glitch tokens within the entire token set, making it a pivotal metric.

Drawing a parallel between K-means clustering and GlitchHunter, it becomes evident that both methodologies are adept at detecting glitch tokens. However, the distinction in their performance is noteworthy. GlitchHunter, in its evaluation, consistently achieves a precision of 100.00% in seven out of the eight models tested and reaches a precision of 99.44% in average, highlighting its reliability. Additionally, it registers an average recall of 63.20% across all models, which is commendable.

On the other hand, while K-means has its merits, its performance is occasionally inconsistent. One primary reason for this inconsistency lies in the K-means algorithm itself. The algorithm's effectiveness is heavily contingent on the initial selection of k center points. This inherent dependency can introduce a degree of randomness into its results. Such unpredictability, in turn, can jeopardize the stability of the algorithm, making it susceptible to occasional inaccuracies, especially when pinpointing glitch tokens. In a domain where precision is paramount, such variability could be a potential limitation.

Delving into ChatGLM2-6b, we analyze the false positive tokens flagged by GlitchHunter. When contrasting with ChatGLM-6b, a striking observation emerges: the vocabulary size of ChatGLM2-6b has shrunk by 50%, plummeting from 130,344 to 64,794. Intriguingly, the dimension of the embedding space remains unchanged, resulting in a denser distribution of tokens. This heightened density adversely impacts the precision of GlitchHunter. A noteworthy side effect of this reduced vocabulary size is the substantial increase in the number of glitch tokens, which surge from 551.40 in ChatGLM-6b to 3,677.40 in ChatGLM2-6b.

7 THREATS TO VALIDITY

Internally, our primary concerns revolve around the potential biases in manual annotations and the selection of configurable options during design. For the annotations, we engage three experienced authors to independently annotate the data, seeking to reduce individual biases. As for the design options, they are set empirically. The specific values used in our experiments are disclosed on our website [15]. While these decisions might influence GlitchHunter's efficacy, our results indicate that, given the current option values, GlitchHunter surpasses established techniques. Optimization of these options remains an area for future exploration.

Externally, the threats pertain to our experimental framework. Due to the inherent unpredictability of the LLMs we assessed, we've instituted controls. By fixing the temperature at 0 and replicating each experiment five times, we aim to curtail variability. Subsequent statistical tests further validate our results. To ensure comprehensive applicability, we utilize a diverse dataset, featuring eight LLMs of varying dimensions and attributes.

8 DISCUSSIONS

8.1 Glitch Token Taxonomy

In this work, we adopt a bottom-up approach to formulate a taxonomy. This methodology enables us to construct categories grounded in the tangible phenomena and specific attributes inherent to the tokens. By categorizing from specific instances to broader classifications, we gain granular insights into the intricacies of each glitch token type. However, it's worth noting that our bottom-up perspective is just one lens through which we can view this domain. A contrasting top-down approach, where one commences with overarching categories and subsequently drills down to specifics, might also be valuable. Such an approach could unveil different, or perhaps even complementary, perspectives on glitch token behaviors and their manifestations. Therefore, the potential interplay and synergy between these two methodologies could be instrumental in achieving a holistic and nuanced understanding of glitch anomalies in LLMs. Future work might consider juxtaposing these methods to uncover any converging or diverging findings, enriching our grasp of the glitch token landscape.

8.2 Glitch Token Detection

The process of detecting glitch tokens in LLMs is central to preserving their consistency, reliability, and overall utility in various applications. As LLMs solidify their position at the forefront of computational linguistics and natural language processing, the presence of glitch tokens becomes a significant concern. These tokens, if unaddressed, can manifest in a myriad of ways, ranging from subtle deviations in model predictions to overtly incorrect or misleading outputs. Such anomalies can compromise the perceived reliability of LLMs, diminishing user confidence and hampering the potential benefits they bring to diverse applications. Beyond immediate concerns, glitch tokens pose a latent threat by introducing vulnerabilities that can be exploited or manipulated. Hence, by focusing on the early detection and rectification of these tokens, we not only reinforce the

robustness of existing LLMs, but also lay the groundwork for improved model integrity in the future. This proactive approach to glitch token management exemplifies the commitment to maintain high standards of quality and reliability, ensuring that as LLMs evolve, they continue to deliver accurate and trustworthy results.

8.3 Glitch Token Mitigation

The mitigation of glitch tokens in LLMs stands as a pivotal aspect of ensuring both the quality and dependability of their outputs. As LLMs increasingly underpin a variety of computational applications, from automated chatbots to sophisticated natural language processors, the inadvertent influence of glitch tokens can precipitate unintended semantic shifts, leading to potential inaccuracies or misrepresentations. To counteract these challenges, a multifaceted approach to mitigation becomes indispensable. This approach encompasses the meticulous curation and refinement of training datasets, the advancement of model architectures to be more discerning, and the implementation of rigorous post-training evaluations. Such evaluations emphasize the detection and rectification of anomalies that can be attributed to glitch tokens. Furthermore, by integrating feedback loops and continuous learning mechanisms, we can ensure that models adapt and evolve in response to newly identified glitches. Through these proactive mitigation measures, we not only enhance the robustness of LLMs against unforeseen disruptions but also establish a benchmark for maintaining the integrity and consistency of language models in diverse operational environments.

9 RELATED WORK

In the related work section, we discuss key areas closely related to our investigation. The emphasis lies on LLM and deep learning model testing [23, 38, 42, 45, 46, 50, 53]. Specifically, we spotlight testing techniques applied to language models [9, 14, 16, 31, 44, 47, 48]. A crucial aspect of this is the occurrence of glitch tokens. We underscore their importance in model testing. Our focused discussion provides a clear context for our study.

9.1 Deep Learning Model Testing

Deep learning model testing is at the forefront of numerous studies. Named Entity Recognition (NER) systems, enhanced by deep neural networks, are pivotal for tasks like sentiment analysis, but their intricacies can lead to errors, such as misclassifying female names as chemicals. The TIN [50] technique addresses this, ensuring consistent NER outputs and demonstrating high precision across various models and APIs. Meanwhile, as social media's prevalence rises, moderating toxic content becomes crucial. While current tools are effective, they often miss malevolent inputs hidden as text in images. The OASIS [45] framework tackles this, creating challenging test cases and identifying errors in major moderation softwares. Furthermore, widespread conversational AI systems like ChatGPT and Siri exhibit biases, prompting the introduction of BiasAsker [42]. This tool, leveraging a comprehensive bias dataset, pinpoints biases in key conversational platforms. Yet, GlitchHunter uniquely focuses on evolving software within LLMs.

9.2 Language Model Testing

The domain of content moderation and validation in computational linguistics has witnessed a surge of groundbreaking research [4, 6–8, 18, 24, 25, 43, 49], contributing unique methodologies and insights. Specifically, the study by [44] navigates the multifaceted world of multimedia content moderation. They specifically target platforms like Facebook and TikTok, introducing a novel technique that synthesizes various modalities, producing and then testing against newly constructed toxic content. Concurrently, the MTTM framework presented in [47] shifts the spotlight to the textual intricacies evident in platforms such as Twitter. By leveraging metamorphic testing, it

meticulously crafts test cases, revealing potential vulnerabilities in moderation systems. In the realm of machine translation, PatInv, as delineated in [16], unveils the persistent challenges. It proposes an innovative testing paradigm that scrupulously assesses translations for both syntactic and semantic fidelity. While these contributions have indisputably enriched content moderation and translation methodologies, our research trajectory distinctly focuses on the exploration, identification, and comprehensive understanding of glitch tokens present in Large Language Models.

9.3 Glitch Token Phenomenon

The glitch token phenomenon in Large Language Models (LLMs) has garnered attention from various pioneers, leading to a series of insightful discoveries. Initially, the work presented in [1] shines light on an intriguing observation: certain tokens exhibited a tendency to aggregate in the semantic landscape of LLMs. Building on this, the research in [37] embarks on a systematic exploration using a repertoire of repetitive prompts, designed to uncover the mysteries of these glitch tokens. An intriguing discovery emerges in [33], where a specific glitch token, termed "petertodd", is brought into focus. This token is subjected to an in-depth analysis, spanning multiple dimensions such as word properties, poetry interpretations, storytelling nuances, and cultural implications. Advancing the discourse, the investigation in [39] delves into categorizing glitch tokens within Text-Davinci-003, while also attempting to trace their semantic footprints across the digital realm. Lastly, the study in [13] widens the net to capture more elusive glitch tokens present in GPT-3.5-turbo and GPT-4. By leveraging the 'Repeat after me:' prompt as a detection mechanism, this research enriches our understanding by correlating the prevalence of glitch tokens with their positions, as represented by token indices.

10 CONCLUSION

In this work, we embark on an empirical analysis to delve deep into the intricacies of glitch tokens in LLMs. Analyzing 7,895 real-world glitch tokens from seven renowned LLMs and three commercial counterparts, we seek to comprehend their manifestations and classifications. This analysis yields pivotal insights, paving the way for enhanced glitch token diagnosis and setting the trajectory for future investigations in this domain. Leveraging these findings, we introduce an innovative approach utilizing clustering in the embedding space to pinpoint glitch tokens, complemented by a token embedding graph to depict token relationships within that space. This culminates in the creation of our glitch token detection mechanism, GlitchHunter, tailored to identify glitch tokens in LLMs seamlessly. Preliminary tests of GlitchHunter on eight expansive LLMs attest to its efficacy and efficiency in glitch token detection. As we look ahead, we aim to amplify GlitchHunter's detection prowess by unearthing novel glitch token characteristics in predominant LLMs and strategizing on glitch token mitigation to bolster LLM reliability and resilience.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for improving this manuscript. This research has been supported by the National Natural Science Foundation of China (grant No.62302176). This research has also been partially supported by NTU College of Engineering CRP and Tier 3 Preparatory Grant 2023.

REFERENCES

[1] (Accessed on 09/25/2023). SolidGoldMagikarp (plus, prompt generation). https://www.lesswrong.com/posts/aPeJE8bSo6rAFoLqg/solidgoldmagikarp-plus-prompt-generation.

[2] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (oct 2008), P10008. https://doi.org/10.1088/1742-5468/2008/10/P10008

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[4] Zhiyuan Chang, Mingyang Li, Yi Liu, Junjie Wang, Qing Wang, and Yang Liu. 2024. Play Guessing Game with LLM: Indirect Jailbreak Attack with Implicit Clues. *arXiv preprint arXiv:2402.09091* (2024).

[5] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E Gonzalez, et al. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality. *See https://vicuna. lmsys. org (accessed 14 April 2023)* (2023).

[6] Gelei Deng, Yi Liu, Yuekang Li, Kailong Wang, Ying Zhang, Zefeng Li, Haoyu Wang, Tianwei Zhang, and Yang Liu. 2024. MASTERKEY: Automated jailbreaking of large language model chatbots. In *NDSS*.

[7] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. 2023. Pentestgpt: An llm-empowered automatic penetration testing tool. *arXiv preprint arXiv:2308.06782* (2023).

[8] Gelei Deng, Yi Liu, Kailong Wang, Yuekang Li, Tianwei Zhang, and Yang Liu. 2024. Pandora: Jailbreak GPTs by Retrieval Augmented Generation Poisoning. *NDSS AISCC* (2024).

[9] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. *arXiv preprint arXiv:2304.02014* (2023).

[10] Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2022. GLM: General Language Model Pretraining with Autoregressive Blank Infilling. *ACL*, 320–335.

[11] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, Vol. 96. 226–231.

[12] U. Flick. 2009. *An Introduction to Qualitative Research*. SAGE Publications. https://books.google.com.sg/books?id=sFv1oWX2DoEC

[13] A Search for More ChatGPT / GPT-3.5 / GPT-4 "Unspeakable" Glitch Tokens. (Accessed on 09/26/2023). https://www.lesswrong.com/posts/kmWrwtGE9B9hpbgRT/a-search-for-more-chatgpt-gpt-3-5-gpt-4-unspeakable-glitch.

[14] Aidan Gilson, Conrad W Safranek, Thomas Huang, Vimig Socrates, Ling Chi, Richard Andrew Taylor, David Chartash, et al. 2023. How does ChatGPT perform on the United States medical licensing examination? The implications of large language models for medical education and knowledge assessment. *JMIR Medical Education* 9, 1 (2023), e45312.

[15] GlitchHunter. (Accessed on 03/05/2024). https://sites.google.com/view/glitchhunter-fse2024.

[16] Shashij Gupta, Pinjia He, Clara Meister, and Zhendong Su. 2020. Machine Translation Testing via Pathological Invariance. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 863–875. https://doi.org/10.1145/3368089.3409756

[17] J Hartigan. 1975. The K-means algorithm. *Clustering algorithms* 4 (1975).

[18] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An empirical study on fine-tuning large language models of code for automated program repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1162–1174.

[19] Hierarchical Clustering in Machine Learning. (Accessed on 09/27/2023). https://www.geeksforgeeks.org/ml-hierarchical-clustering-agglomerative-and-divisive-clustering/.

[20] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. arXiv:2310.06825 [cs.CL]

[21] James Melville Leland McInnes, John Healy. 2018. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. arXiv:1802.03426 [stats.ML]

[22] Stephanie Lin, Jacob Hilton, and Owain Evans. 2022. Teaching Models to Express Their Uncertainty in Words. arXiv:2205.14334 [cs.CL]

[23] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 530–543.

[24] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, and Yang Liu. 2023. Prompt Injection attack against LLM-integrated Applications. *arXiv preprint arXiv:2306.05499* (2023).

[25] Yi Liu, Gelei Deng, Zhengzi Xu, Yuekang Li, Yaowen Zheng, Ying Zhang, Lida Zhao, Tianwei Zhang, and Yang Liu. 2023. Jailbreaking chatgpt via prompt engineering: An empirical study. *arXiv preprint arXiv:2305.13860* (2023).

[26] Edward Loper and Steven Bird. 2002. NLTK: The Natural Language Toolkit. arXiv:cs/0205028 [cs.CL]

[27] ML | K means++ Algorithm. (Accessed on 09/27/2023). https://www.geeksforgeeks.org/ml-k-means-algorithm/.

[28] C. Model card Models and evaluations for claude models. (Accessed on 09/25/2023). https://www-files.anthropic.com/production/images/Model-Card-Claude-2.pdf.

[29] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, Johannes Heidecke, Pranav Shyam, Boris Power, Tyna Eloundou Nekoul, Girish Sastry, Gretchen Krueger, David Schnurr, Felipe Petroski Such, Kenny Hsu, Madeleine Thompson, Tabarak Khan, Toki Sherbakov, Joanne Jang, Peter Welinder, and Lilian Weng. 2022. Text and Code Embeddings by Contrastive Pre-Training. arXiv:2201.10005 [cs.CL]

[30] Ann E. Nicholson, Kevin B. Korb, Erik P. Nyberg, Michael Wybrow, Ingrid Zukerman, Steven Mascaro, Shreshth Thakur, Abraham Oshni Alvandi, Jeff Riley, Ross Pearson, Shane Morris, Matthieu Herrmann, A.K.M. Azad, Fergus Bolger, Ulrike Hahn, and David Lagnado. 2020. BARD: A structured technique for group elicitation of Bayesian networks to support analytic reasoning. arXiv:2003.01207 [cs.AI]

[31] Hyunjong Ok and Seong-Bae Park. 2023. Post-Trained Language Model Adaptive to Extractive Summarization of Long Spoken Documents. In *ICASSP 2023-2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 1–2.

[32] OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[33] The petertodd phenomenon. (Accessed on 09/25/2023). https://www.lesswrong.com/posts/jkY6QdCfAXHJk3kea/the-petertodd-phenomenon.

[34] ShareGPT52K. (Accessed on 03/06/2024). https://huggingface.co/datasets/RyokoAI/ShareGPT52K.

[35] Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, Gretchen Krueger, Jong Wook Kim, Sarah Kreps, Miles McCain, Alex Newhouse, Jason Blazakis, Kris McGuffie, and Jasmine Wang. 2019. Release Strategies and the Social Impacts of Language Models. arXiv:1908.09203 [cs.CL]

[36] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.

[37] SolidGoldMagikarp II: technical details and more recent findings. (Accessed on 09/25/2023). https://www.lesswrong.com/posts/Ya9LzwEbfaAMY8ABo/solidgoldmagikarp-ii-technical-details-and-more-recent.

[38] Archana Tikayat Ray, Olivia J Pinon-Fischer, Dimitri N Mavris, Ryan T White, and Bjorn F Cole. 2023. aeroBERT-NER: Named-Entity Recognition for Aerospace Requirements Engineering using BERT. In *AIAA SCITECH 2023 Forum*. 2583.

[39] SolidGoldMagikarp III: Glitch token archaeology — LessWrong. (Accessed on 09/26/2023). https://www.lesswrong.com/posts/8viQEp8KBg2QSW4Yc/solidgoldmagikarp-iii-glitch-token-archaeology.

[40] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[41] Nees Jan van Eck Vincent Traag, Ludo Waltman. 2018. From Louvain to Leiden: guaranteeing well-connected communities. arXiv:1810.08473 [cs.SI]

[42] Yuxuan Wan, Wenxuan Wang, Pinjia He, Jiazhen Gu, Haonan Bai, and Michael Lyu. 2023. BiasAsker: Measuring the Bias in Conversational AI System. arXiv:2305.12434 [cs.CL]

[43] Guanyu Wang, Yuekang Li, Yi Liu, Gelei Deng, Tianlin Li, Guosheng Xu, Yang Liu, Haoyu Wang, and Kailong Wang. 2024. MeTMaP: Metamorphic Testing for Detecting False Vector Matching Problems in LLM Augmented Generation. *FORGE* (2024).

[44] Wenxuan Wang, Jingyuan Huang, Chang Chen, Jiazhen Gu, Jianping Zhang, Weibin Wu, Pinjia He, and Michael Lyu. 2023. Validating Multimedia Content Moderation Software via Semantic Fusion. arXiv:2305.13623 [cs.SE]

[45] Wenxuan Wang, Jingyuan Huang, Jen tse Huang, Chang Chen, Jiazhen Gu, Pinjia He, and Michael R. Lyu. 2023. An Image is Worth a Thousand Toxic Words: A Metamorphic Testing Framework for Content Moderation Software. arXiv:2308.09810 [cs.SE]

[46] Wenxuan Wang, Jen-tse Huang, Weibin Wu, Jianping Zhang, Yizhan Huang, Shuqing Li, Pinjia He, and Michael R Lyu. 2023. Mttm: Metamorphic testing for textual content moderation software. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2387–2399.

[47] Wenxuan Wang, Jen tse Huang, Weibin Wu, Jianping Zhang, Yizhan Huang, Shuqing Li, Pinjia He, and Michael Lyu. 2023. MTTM: Metamorphic Testing for Textual Content Moderation Software. arXiv:2302.05706 [cs.CL]
[48] Tianwen Wei, Jian Luan, Wei Liu, Shuang Dong, and Bin Wang. 2023. CMATH: Can Your Language Model Pass Chinese Elementary School Math Test? *arXiv preprint arXiv:2306.16636* (2023).
[49] Zihao Xu, Yi Liu, Gelei Deng, Yuekang Li, and Stjepan Picek. 2024. LLM Jailbreak Attack versus Defense Techniques–A Comprehensive Study. *arXiv preprint arXiv:2402.13457* (2024).
[50] Boxi Yu, Yiyan Hu, Qiuyang Mang, Wenhan Hu, and Pinjia He. 2023. Automated Testing and Improvement of Named Entity Recognition Systems. arXiv:2308.07937 [cs.CL]
[51] Shyr-Shen Yu, Shao-Wei Chu, Ching-Lin Wang, Yung-Kuan Chan, and Chia Yi Chuang. 2014/05. A Modified K-means Algorithms - Bi-Level K-Means Algorithm. In *Proceedings of the 2nd International Conference on Soft Computing in Information Communication Technology*. Atlantis Press, 10–13. https://doi.org/10.2991/scict-14.2014.3
[52] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. 2022. Glm-130b: An open bilingual pre-trained model. *arXiv preprint arXiv:2210.02414* (2022).
[53] Zhen Zhang, Mengting Hu, Shiwan Zhaofor, Minlie Huang, Haotian Wang, Lemao Liu, Zhirui Zhang, Zhe Liu, and Bingzhe Wu. 2023. E-NER: Evidential Deep Learning for Trustworthy Named Entity Recognition. arXiv:2305.17854 [cs.CL]
[54] Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. 2020. Fine-Tuning Language Models from Human Preferences. arXiv:1909.08593 [cs.CL]

Received 2023-09-28; accepted 2024-04-16