Enhancing Language Model with Unit Test Techniques for Efficient Regular Expression Generation

Chenhui Mao, Xiexiong Lin, Xin Jin, Xin Zhang
Ant Group
{maochenhui.maochen, xiexiong.lxx, king.jx, evan.zx}@antgroup.com

Abstract

Recent research has investigated the use of generative language models to produce regular expressions with semantic-based approaches. However, these approaches have shown shortcomings in practical applications, particularly in terms of *functional correctness*, which refers to the ability to reproduce the intended function inputs by the user. To address this issue, we present a novel method called Unit-Test Driven Reinforcement Learning (UTD-RL). Our approach differs from previous methods by taking into account the crucial aspect of functional correctness and transforming it into a differentiable gradient feedback using policy gradient techniques. In which functional correctness can be evaluated through Unit Test, a testing method that ensures regular expressions meets its design and performs as intended. Experiments conducted on public datasets demonstrate the effectiveness of the proposed method in generating regular expressions. This method has been employed in a regulatory scenario where regular expressions can be utilized to ensure that all online content is free from non-compliant elements, thereby significantly reducing the workload of relevant personnel.

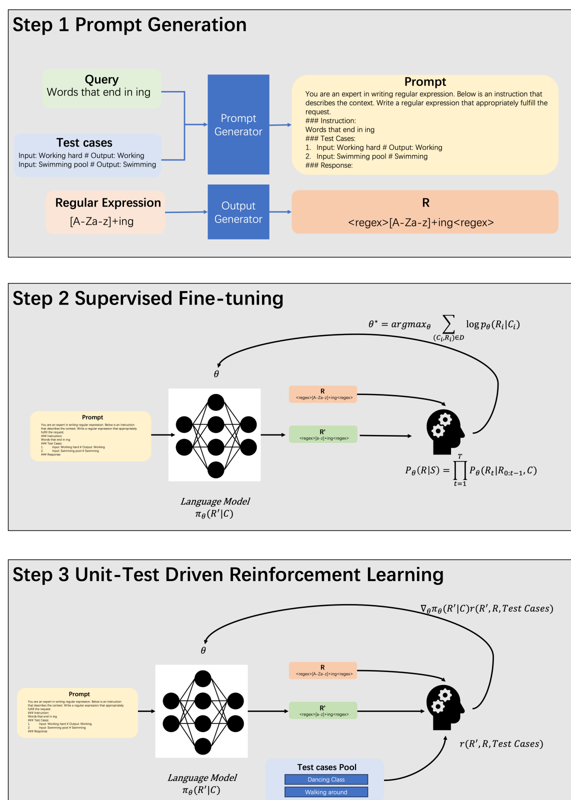

Figure 1: Pipeline of our works. And the whole pipeline consists of 3 steps: the first step will generate prompt from the original context; followed by the SFT with the prompt generated from the first step; finally Unit-Test Driven Reinforcement Learning is implemented

1 Introduction

Regular expressions are an essential tool for processing text in an efficient, flexible, and powerful manner (Friedl, 2006). For instance, an individual whose work involves reviewing the language used in an application to prevent the display of violent or pornographic content to underage users. Manually checking each line can be a time-consuming task. Therefore, the use of regular expressions can greatly streamline this process. Nevertheless, writing and debugging regular expressions can be a daunting task for those without expertise, as the syntax can often be obscure and unintuitive (Karttunen et al., 1996).

The use of natural language to generate regular expressions has been explored in several works to bridge the gap for the public in utilizing regular expressions. For instance, Ranta et al. (Ranta, 1998) developed a rule-based system that generates regular expressions from template input. Subsequently, Locascio et al. (Locascio et al., 2016) proposed the use of LSTM-based Sequence to Sequence models to generate regular expressions based on contextual inputs. Furthermore, with the advancement of large language models, researchers have discovered that the performance can be improved by employing Supervised Fine-tuning (SFT) (Ouyang et al., 2022) on Large Language Models (LLMs). Nev-

Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: Industry Track, pages 12–19
December 6-10, 2023 ©2023 Association for Computational Linguistics

ertheless, regular expressions generated by these models often encounter compilation failures and inadequately capture the intended functionality of the input requirements, which is a critical aspect in practical applications. To address this, researchers have explored the use of semantic correctness (Park et al., 2019) as a criterion. However, adopting such a method does not completely resolve the aforementioned issues. We posit that the disregard for functional significance in input specification may be a significant factor contributing to these challenges.

Therefore, this paper emphasizes the importance of functional correctness. To enhance the functional correctness of the generated regular expression, it is important to consider the practical context in which it will be used. Generally, assessing its practical applicability requires conducting "Unit Test". Specifically, if the generated regular expression can accurately extract the desired results from a given sequence of inputs, it can be considered to meet the functional requirements of the user. Therefore, in this paper, we propose **U**nit-**T**est **D**riven **R**einforcement **L**earning (UTD-RL). This approach utilizes policy gradient techniques (Sutton et al., 1999) to learn from the feedback provided by the unit test results, enabling the model to adjust its pattern generation process to better align with the intended functionality. As a result, it shows promise in improving the effectiveness of regular expression generation in practical applications. Experimental results demonstrate that regular expressions generated by this method can better adhere to the input requirements, resulting in a significant improvement in the performance of the generated regular expression with respect to Unit Test.

As mentioned earlier, we consider functional correctness to be the most crucial factor in this task. However, we have observed that the previous evaluation method, which computes equivalence by converting each regular expression to a minimal deterministic finite automaton (DFA) and leveraging the fact that minimal DFAs are guaranteed to be the same for semantically equivalent regular expressions, is inadequate for assessing the functional correctness of the generated regular expression in relation to the input requirements. Therefore, in this paper, we propose the adoption of "Unit Test" as an alternative method for evaluating the generated regular expression, in addition to utilizing DFA.

To sum up our contributions:

1. we came up with the **UTD-RL** approach that utilizes the outcomes of "Unit Test" to enhance the functional correctness of the generated regular expression in alignment with input specifications.

2. we propose the use of "**Unit Test**" for evaluation, as it can better reflect the degree of fulfillment of the input requirements.

3. we conducted several experiments to validate the efficacy of the **UTD-RL** approach.

2 Related Work

Recent research has focused on automating the generation of regular expressions from natural language, employing both non-deep learning and deep learning approaches. Early researchers highlighted the ability to encode regular expressions into finite state networks (Karttunen et al., 1996). Ranta et al. (Ranta, 1998) capitalized on this property and developed a rule-based technique for converting formatted language specifications into regular expressions. Sequentially, Locascio et al. (Locascio et al., 2016) first introduced an LSTM-based sequence-to-sequence model (Deep Regex) that translates contextual information into regular expressions using a syntax-based objective: maximum likelihood estimation (MLE). Zhong and Bhatia (Zhong et al., 2018) optimized performance by employing policy gradient techniques (Sutton et al., 1999)to train the model with a semantics-based objective. Similarly, Park et al. (Park et al., 2019) applied semantic correctness as the reinforcement learning reward. However, experiments conducted on these models revealed significant overfitting on public datasets resulting in limited generalizability to other input requirements. We speculate that LSTM lacking the capacity for induction and deduction compared to the advanced large language models available today.

Recently, Large language models (LLMs) trained on extensive text corpora from diverse domains have exhibited their capability to perform zero-shot tasks, including code generation. This zero-shot ability emerged when models reached an adequate scale (Brown et al., 2020). Researchers utilizing pre-trained LLMs and fine-tuning them on pertinent datasets have achieved remarkable outcomes. For example, CodeX (Chen et al., 2021),

a fine-tuned model on GPT-3(Brown et al., 2020), outperforms prior state-of-the-art models on code generation. Copilot, a highly renowned code suggestion tool within the GitHub community, employs CodeX as its foundational model. Furthermore, CodeGeeX (Zheng et al., 2023), a multilingual code generation model equipped with 13 billion parameters, attains the highest average performance on publicly available datasets.

3 Methods

3.1 Language Model

We conducted experiments on large language models, such as llama, GPT-3, and text-davinci-003, to evaluate their performance in solving public regular expression problems. The results demonstrate their ability to generate regular expressions, although their performance may not be on par with prior research advancements on public datasets. This finding is significant, particularly because these models are pretrained on a vast corpus rather than being specifically designed for regular expression generation. Consequently, it is essential to fine-tune these language models specifically for the task of regular expression generation to improve their effectiveness.

3.2 Unit-Test Driven Reinforcement Learning

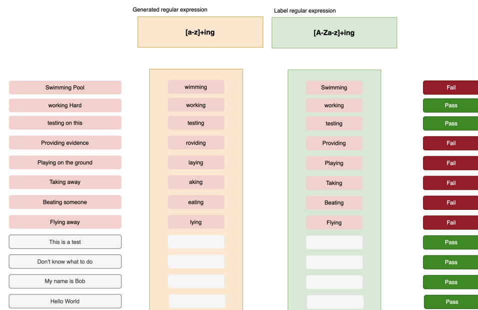

Figure 2: Unit test. Unit test are conducted on both the generated regular expression and the target regular expression. If the extracted outcome is the same, the test case is considered passed. Otherwise, the test case fails.

Ensuring functional correctness is a critical aspect of regular expressions. To clarify, in practical applications, validating the correctness of a regular expression usually involves unit test. If all the intended patterns are successfully extracted from the test cases and all of the extracted patterns match the desired patterns, then the regular expression is considered valid. Unfortunately, previous researches employing SFT on language models overlooked this aspect. As a solution, we propose utilizing policy gradient method (Sutton et al., 1999), which optimizes parameterized policies through gradient descent based on the expected return (reward) to convert functional correctness into a differentiable gradient.

Our approach aims to improve the functional correctness of the model by highlighting the unique functionality of regular expressions and encouraging the production of functionally correct regular expressions, especially in challenging scenarios where the generation process failed to compile. The reinforcement phase will facilitate the model in learning to generate regular expressions that are both semantically and functionally correct, leading to improved performance on "Unit Test". Specifically, for a given problem context C_i, a desired ground truth regular expression R_i and several valid test cases T_i, we want to maximize the expected reward $r(y, R_i, T_i)$ for every regular expression y generated by language model p_θ, namely improving the ratio of the generated regular expression y that can pass the unit test.

$$J(\theta) = \sum_{(C_i, R_i, T_i) \in \mathbb{D}} E_{y \sim p_\theta(\cdot|C_i)} r(y, R_i, T_i) \quad (1)$$

During the training process, it is still desirable for the regular expressions generated by the model to have a minimal discrepancy with ground truth annotated regular expressions. Therefore, we incorporate the supervise loss with the ground truth regular expressions into the final objective function, aiming to mitigate the disparity.

$$obj(\theta) = \beta J(\theta) + \gamma E_{C \sim \mathbb{D}} \log p_\theta(y|C) \quad (2)$$

In this context, $\mathbb{D}$ is a regular expression problem set. The reward coefficient, β, and supervise loss coefficient, γ, control the magnitude of importance between the reward and the supervise loss. Setting γ to 0 would make the gradient depend solely on the functional correctness of the generated regular expression.

Measurement of Functional Correctness. Since we have utilized the policy gradient method (Sutton et al., 1999) to transform functional correctness into a differentiable signal, it is crucial to define a criterion for evaluating functional correctness. In practical terms, a regular expression is considered valid if it can successfully extract the

desired string pattern from a provided set of inputs. This concept shares similarities with the pass@k metric employed in code evaluation (Chen et al., 2021). To accomplish this, we employ dedicated unit test designed for regular expressions to assess their functional correctness. These unit test, specifically tailored to regular expressions, are illustrated in Figure 2. The pseudo code in Algorithm 1 illustrates the process of the reward function. If a generated regular expression passes the current test case t_j, a positive value is added to the reward. Otherwise, a negative value is added to the reward.

Algorithm 1: Reward Function

```
input  : Label Regex R_i &
         Predicted Regex y &
         Test_Cases T_i = {t_1, ..., t_n}
output : r
1  r ← 0;
2  Initialize On p & n;
3  for t_j ∈ T_i do
4      if y Fail the compilation then
5          r ← 0 − n;
6          continue
7      end
8      str_1 ← Pattern_Match(t_j, y);
9      str_2 ← Pattern_Match(t_j, R_i);
10     if str_1==str_2 then
11         r ← r + p
12     else
13         r ← r − n
14     end
15 end
```

Test Case Generation. Generating appropriate test cases is a crucial aspect of unit test. Although manual generation is possible, it is often unnecessary due to the availability of automated tools like **rstr**, which can generate test cases automatically based on the provided regular expression. For thorough testing, it is essential to include both positive test cases, denoted as $\{t_i^+\}$, which match the regular expression pattern, and negative test cases, denoted as $\{t_i^-\}$, which do not produce any matches. Accordingly, we define our set of test cases as $T_i = \{t_1^+, t_2^+, ..., t_1^-, t_2^-, ...\}$, comprising positive cases generated using **rstr** and negative cases randomly selected from pre-generated test case pools.

id	regular expression
1	^[1-9]\d * $
2	^([1-9][0-9]*){1,3}$
3	^\+?[1-9][0-9]*$

Table 1: Example on regular expression A common regular expression problem that can be found on StackOverflow: match non-zero positive integer

3.3 Evaluation

DFA Equivalence. We assessed the effectiveness of our approach in generating regular expressions by testing it with DFA Equivalence, a method that converts a given regular expression into a minimal DFA. As noted by Karttunen (1996)(Karttunen et al., 1996), regular expressions can be represented by finite state networks. This approach is grounded in the fact that two equivalent regular expressions possess identical minimal DFAs, irrespective of their structural dissimilarities(Hopcroft et al., 2001).

However, DFA Equivalence falls short when dealing with large and complex regular expressions. While DFA Equivalence converts a regular expression into a Deterministic Finite Automaton, its primary focus is on syntactical equivalence between the generated regular expression and the reference solution. However, functionally equivalent regular expression may have syntactically different forms. For example, the regular expressions in Table 1 capture the pattern of non-zero positive integers; nevertheless, DFA Equivalence fails to identify these regular expressions as representing the same input specification. This limitation is especially significant in complex real-world scenarios where different experts might create distinct regular expressions for the same specification.

Unit Test. In Section 3.2, we introduced the use of unit test to capture functional correctness during the reinforcement learning process. At the evaluation stage, this technique can be employed to assess the functional correctness of the generated regular expression. For better clarity, we have created a dedicated test case pool for each regular expression problem, as depicted in Figure 2. The problem is considered solved only if the generated regular expression passes all the test cases. Therefore we can define the metric as the number of solved regular expression problems out of the total numbers.

$$pass_i = \begin{cases} 1 & \text{if pass all test cases} \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

$$Unit\ Test = \frac{\sum_i \mathbb{1}\{pass_i = 1\}}{\sum_i 1} \quad (4)$$

4 Experimental Setup

In this section, we evaluate our work on different pre-trained language models to verify its effectiveness. Additionally, we conduct test case analysis and present case studies to provide further insights.

4.1 Model Configuration

We conducted experiments to evaluate the effectiveness of UTD-RL on large language models: GPT-3 (Brown et al., 2020) and LLaMA (Touvron et al., 2023). The pretrained GPT-3 models were provided by ModelsScope [1], a platform developed by the Alibaba DAMO team. The pretrained LLaMA weights can be found on Hugging Face [2].

4.2 Reinforcement Learning Setup

We perform a hyper-parameter search to determine the best hyper-parameters: β and γ were set to 0.01 and 1.0, respectively. The number of test cases was set to 10. Out of these test cases, 9 were derived from positive cases, and 1 was derived from a negative case.

4.3 Dataset

Our experiments are conducted on the following datasets.

NL-RX-Pub. A merge dataset from KB13 (Kushman and Barzilay, 2013), NL-RX-Synth (Locascio et al., 2016) and NL-RX-Turk(Locascio et al., 2016). The pairs are divided into three subsets: a 65% training set, a 10% development set, and a 25% testing set (testing set are divided back into KB13, NL-RX-Synth, NL-RX-Turk accordingly). In order to avoid data leakage problem, the division is followed by the target regular expression.

NL-RX-ST[3], In order to test the generalizability on public regular expression problems, we manually mount 100 regular expression problems from public resources including but not limited to github, wikipedia, and stackoverflow. To be noted this dataset should only be used for testing.

[1] model weight can be found in https://modelscope.cn/models/damo/nlp_gpt3_text-generation_1.3B/summary

[2] model weight can be found in https://huggingface.co/decapoda-research/llama-7b-hf

[3] Dataset available on https://github.com/Morris135212/NL-RX

4.4 Results and Analysis

We demonstrate the effectiveness of our approach by comparing it to the existing approaches including Deep Regex(Locascio et al., 2016) and SoftRegex(Park et al., 2019). Moreover, we fine-tune text-davinci-003 (SFT API provided by OpenAI) on same data. We also conduct the ablation experiments to compare the results obtained from different language models with and without UTD-RL.

Baseline Comparison. Table 2 provides a summary of our results across various methods.

1. **Deep Regex** & **SoftRegex** Both the Deep Regex(Locascio et al., 2016) and SoftRegex(Park et al., 2019) have a simple model structure based on LSTM and utilize a syntax-based objective (MLE) for training. These methods perform well on public datasets, but they exhibit limited generalization ability on unseen problems, as demonstrated by the results on NL-RX-ST. This demonstrates that they severely over-fit on training data. Such a shortcoming stem from the model itself being too simplistic and the insufficient utilization of functional correctness of the generated regular expression. We conducted futher fine-tuning of the training data using a more sophisticated model with an increased number of parameters. The obtained results provide substantial support for our claim.

2. **text-davinci-003** It is widely acknowledged that scaling up language models, such as increasing training compute and model parameters, can significantly improve performance and sample efficiency across various downstream NLP tasks(Wei et al., 2022).Text-davinci-003, as one of the current state-of-the-art large language model provided by openai, shows promising performance across all datasets. It even demonstrates some ability to generalize to unseen problems. However, the model treats the problem as a black box, only leveraging the syntax similarity of regular expressions. Therefore, by better utilizing the inherent functionality of the regular expression, we can further enhance the effectiveness of the model. This point has been proven in subsequent ablation studies.

3. **GPT-3** & **llama**. Both models are currently open-source, large language models. From

Model	UTD-RL	DFA-Acc	Unit Test	DFA-Acc	Unit Test	DFA-Acc	Unit Test	Unit Test
		KB13		NL-RX-Synth		NL-RX-Turk		NL-RX-ST
Deep Regex	/	0.6611	0.6627	0.9180	0.9218	0.6420	0.6535	0.12
SoftRegex	/	0.6621	0.6601	0.9222	0.9233	0.6623	0.6676	0.15
text-davinci-003	/	0.6899	0.7422	0.9043	0.9323	0.6753	0.7191	0.43
GPT-3 1.3B		0.6749	0.6869	0.9230	0.9314	0.6636	0.6864	0.31
GPT-3 1.3B	✓	0.6814	0.7234	0.9219	0.9312	0.6782	0.7119	0.37
GPT-3 2.7B		0.6734	0.6889	0.8959	0.9209	0.6663	0.6884	0.33
GPT-3 2.7B	✓	0.6843	0.7297	0.9307	0.9349	0.6813	0.7221	0.40
llama 7B		0.6764	0.7381	0.8998	0.9278	0.6664	0.708	0.37
llama 7B	✓	0.7534	0.7674	0.9223	0.9481	0.6995	0.7219	0.48
llama 13B		0.7442	0.7409	0.899	0.9398	0.6865	0.7235	0.41
llama 13B	✓	**0.7582**	**0.7789**	**0.9237**	**0.9497**	**0.7097**	**0.7348**	**0.53**

Table 2: The experiment results on different approaches (using DFA accuracy and unit test as metrics)

the results, we find that after basic fine-tuning (without UTD-RL), these baselines demonstrate the ability to approximate the performance exhibited by text-davinci-003. However, it treats the problem as a black box, only utilizing the syntax similarity of regular expressions, which we believe is insufficient for functional corpora like regular expressions. Therefore, a later ablation study will show that considering functional correctness greatly improves the performance not only on public datasets but also in terms of generalization ability.

Ablation study. We conducted comprehensive ablation experiments to evaluate the use of UTD-RL on GPT-1.3B, GPT-2.7B, llama-7B, and llama-13B. Table 2 demonstrated a significant enhancement in overall performance by incorporating UTD-RL. The utilization of UTD-RL resulted in an average improvement of 2.06% in DFA-Acc for KB-13, NL-RX-Synth, and NL-RX-Turk, and 2.27% in Unit-Test. Furthermore, it led to an average improvement of 9% in generalization tests for NL-RX-ST. The most notable experimental results were observed with the llama-13B model when employing the UTD-RL approach. The use of UTD-RL with the llama-13B model exhibited considerable improvements across various datasets, surpassing even the results achieved with the text-davinci-003 model. This demonstrate that considering the functional properties inherent in regular expression can enhance the functional capabilities of the model in generating regular expressions. This approach also promotes the generalization ability of the model, enabling it to generate regular expression that meet the functional requirements of input even for unseen problems.

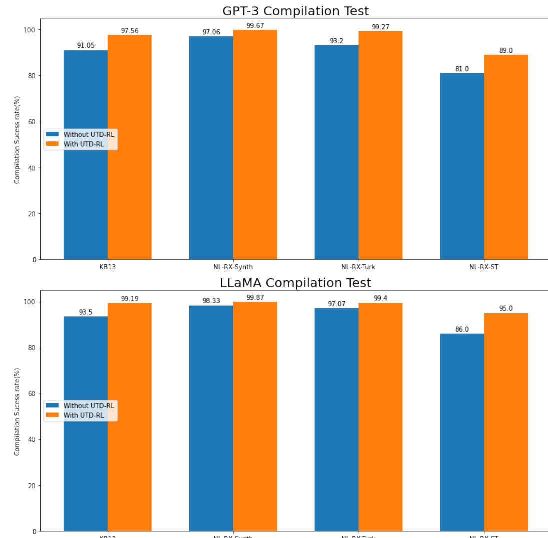

Figure 3: Compilation Test on GPT3 1.3B and LLaMA 7B

Another observation is presented in Figure 3.The use of UTD-RL has resulted in the improved success rates for regular expressions during compilation. Specifically, for gpt-3 1.3B, there were average improvements of 5.06% on KB-13, NL-RX-Synth, and NL-RX-Turk tests, and 8% improvements on NL-RX-ST. For llama-7B, the average improvements were 3.19% on KB-13, NL-RX-Synth, and NL-RX-Turk tests, and 9% improvements on NL-RX-ST. Section 3.2 provides an illustration of the reward function used in UTD-RL, which incorporates a form of "punishment" for generated regular expressions that do not pass compilation. The experimental results support the notion that this reward system enables the model to generate more robust regular expressions.

In conclusion, our method shows great potential for significantly enhancing the functional correctness of natural language-based approaches in

generating regular expressions, In addition, the use of UTD-RL can effectively improve the model's generalization ability in other regular expression problems.

5 Practical application

In our context, the app hosts numerous registered merchants. In compliance with market regulatory requirements, these registered merchants are obligated to undergo internal compliance reviews before publishing new advertisement landing pages or text content. This is done to ensure that the content does not contain any non-compliant elements. Given the large number of merchants involved and the complexity of the rules, the conventional approach relied heavily on manual creation of regular expressions to identify non-compliant text scenarios. For instance, one requirement for advertisement landing pages was the exclusion of promotional expressions. Unfortunately, this approach often resulted in significant time and labor costs associated with the development and testing of regular expressions. Now a new solution has been introduced: an automated workflow that utilizes the large language model trained with UTD-RL. To make it more specific, This language model is capable of generating production-ready regular expressions and automatically conducting unit test, thereby enabling an automated workflow that greatly facilitates the public's use of regular expressions. The process is depicted in Figure 4.

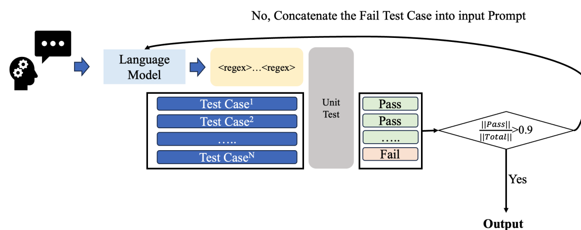

Figure 4: Pipeline for generating a valid regular expression in a practical application. Language model generates a regular expression based on users' requests. Subsequently, a unit test is implemented to assess the validity of the regular expression. If the outcome of the unit test exceeds the threshold, the regular expression is considered valid. Conversely, the input prompt is concatenated with the failed cases to regenerate the regular expression.

6 Conclusion

In conclusion, ensuring the functional correctness of regular expressions is crucial in practical applications. This paper proposes the use of UTD-RL to effectively utilize the outcomes of unit test as rewards for the model, thereby enhancing the functional correctness. Furthermore, "unit test" are employed to assess the functional correctness of the generated regular expressions.

This paper solely focuses on evaluating the effectiveness of the proposed method in the generation of regular expressions. However, it is believed that this approach can be extended to generate any corpus that necessitates functional specifications (e.g., Python code generation, SQL generation, etc.). Future research will investigate the applicability of this method in these domains, and we encourage interested researchers to experiment with this approach.

References

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Jeffrey EF Friedl. 2006. *Mastering regular expressions*. " O'Reilly Media, Inc.".

John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65.

Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schille. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328.

Nate Kushman and Regina Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. North American Chapter of the Association for Computational Linguistics (NAACL).

Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. 2016. Neural generation of regular expressions from natural language with minimal domain knowledge. *arXiv preprint arXiv:1608.03000*.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback.

Jun-U Park, Sang-Ki Ko, Marco Cognetta, and Yo-Sub Han. 2019. Softregex: Generating regex from natural language descriptions using softened regex equivalence. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, pages 6425–6431.

Aarne Ranta. 1998. A multilingual natural-language interface to regular expressions. In *Finite State Methods in Natural Language Processing*.

Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, volume 12. MIT Press.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent abilities of large language models. *Transactions on Machine Learning Research*. Survey Certification.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.

Zexuan Zhong, Jiaqi Guo, Wei Yang, Jian Peng, Tao Xie, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2018. Semregex: A semantics-based approach for generating regular expressions from natural language specifications. In *Proceedings of the 2018 conference on empirical methods in natural language processing*.