Prompt Injection Generation Using Small Language Models with Reinforcement Learning with Artificial Intelligence Feedback

by

Aneesh Gupta

S.B. Computer Science and Engineering and Mathematics, Massachusetts Institute of Technology, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2025

Authored by: Aneesh Gupta
Department of Electrical Engineering and Computer Science
January 31, 2025

Certified by: Amar Gupta
Research Scientist, Thesis Supervisor

Accepted by: Katrina Lacurts
Chair
Masters of Engineering Thesis Committee

Prompt Injection Generation Using Small Language Models with Reinforcement Learning with Artificial Intelligence Feedback

by

Aneesh Gupta

Submitted to the Department of Electrical Engineering and Computer Science
on January 31, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Large language models (LLMs) have become an integral part of many fields from customer support automation to research assistants. However, despite their growing adoption, they face significant challenges, particularly when it comes to safety in sensitive contexts. Existing methods like Reinforcement Learning with Human Feedback (RLHF) and keyword filtering have contributed to improving the robustness of these models, but these approaches are very resource-intensive and the models can still be vulnerable to malicious attacks like prompt injections and jailbreaking. One notable limitation in testing defenses against such attacks is the scarcity of appropriate datasets. This thesis investigates the use of small language models (SLMs) to generate goal hijacking messages, a subset of prompt injection messages. Techniques such as LoRA fine-tuning and full fine-tuning of even smaller models are employed in this short form text generation model. We also introduce a fine-tuned SLM enhanced with Reinforcement Learning with Artificial Intelligence Feedback (RLAIF), which removes reliance on slow human feedback by using faster AI-generated feedback instead. By optimizing the reference model and reward functions, we improve alignment with ground truth prompt injection messages while addressing issues such as mode collapse and overfitting. These findings show promise, and further research is necessary to determine how well the approach can generalize to other domains and perform in real-world scenarios. Future work is likely to focus on multilingual datasets and distributed computation to further extend the applicability and efficiency of the method.

Thesis supervisor: Amar Gupta
Title: Research Scientist

Acknowledgments

I would like to thank Dr. Amar Gupta for his unwavering guidance. From providing me the opportunity to UROP in his lab as a freshman to serving as my MEng supervisor, his technical expertise and leadership have been invaluable throughout my journey at MIT. The opportunities he offered and his mentorship have played a pivotal role in my academic and personal growth. I am also appreciative of the support from Sheila Dada, Lucas Orosco, and others from Itaú Unibanco. I am deeply thankful to the other members of the lab for their collaborative research and insightful advice.

I would like to thank all of my friends. Their presence has provided great motivation and support throughout this endeavour. In particular, I am grateful to my ultimate frisbee teammates for making these past few years truly unforgettable.

Finally, I am profoundly grateful to my family. My parents have always been there for me and my brother has been an outstanding role model. I could not have accomplished any of this without their love and support.

Contents

Title page 1

Abstract 3

Acknowledgments 5

List of Figures 9

List of Tables 11

1 Introduction 13

1.1 Motivation . . . 13

1.2 Problem Statement . . . 14

1.3 Thesis Overview . . . 15

2 Background 17

2.1 Related Work . . . 17

2.1.1 Methods for Generating Prompt Injections . . . 17

2.1.2 Goal Hijacking Techniques . . . 19

2.1.3 Reinforcement Learning for Text Generation . . . 20

2.2 Evaluation Metrics . . . 22

2.2.1 Rouge Score . . . 22

2.2.2 Sentence-BERT Embeddings . . . 23

2.2.3 Mean Dot Score of Embeddings 23

3 Data Preprocessing **25**

3.1 Mosscap Dataset 25

3.2 Dataset Filtering and Cleaning 29

3.3 Custom Prompt Generation 31

4 Prompt Generation SLM Pipeline **35**

4.1 Training Environment Overview 35

4.2 Mistral 7B Parameter Model 36

4.2.1 Training Acceleration Techniques 37

4.2.2 Epoch Count Reduction 39

4.2.3 Effect of Temperature on Model Performance 40

4.3 GPT-2 Large (774M Parameters) Analysis 41

4.4 Reinforcement Learning with Artificial Intelligence Feedback (RLAIF) Pipeline 45

4.4.1 Comparison of Pretrained and Fine-tuned Models 50

5 Discussion **55**

5.1 Approach Comparisons 55

5.2 Future Work 56

6 Conclusion **59**

References **61**

List of Figures

3.1 Distribution of Prompt Injection Messages per Level 26
3.2 Distribution of Prompt Injection Messages by Word Count 27
3.3 Distribution of Filtered Messages by Word Count 30

4.1 2D UMAP Embeddings of Mistral 7B vs Validation Ground Truth 37
4.2 Two-bit K-means Quantization of 2D Matrix 38
4.3 Training Loss for Mistral 7B Model 40
4.4 2D UMAP Embeddings of 0.7 vs 0 temperature 41
4.5 Breakdown of GPT-2 Large Layers 43
4.6 2D UMAP Embeddings of GPT-2 Large vs Validation Ground Truth 45
4.7 RLAIF Pipeline with Reference Model Discriminator 47
4.8 2D UMAP Embeddings of RLAIF from Pretrained GPT-2 Large 50
4.9 2D UMAP Embeddings of RLAIF from Fine-tuned GPT-2 Large 51
4.10 Reward over Number of Steps for Both RLAIF Approaches 52

5.1 RLAIF Pipeline with Mixture of Expert Feedback 58

List of Tables

3.1 Distribution of Word Count Spread Analysis 27

3.2 Distribution of Word Count Shape Analysis 28

3.3 Ten Most Common 3 Word Starts 29

3.4 Comparison of Original and Filtered Dataset Spread 31

3.5 Comparison of Original and Filtered Dataset Shape 31

5.1 Model Rouge and Embedding Dot Score Results 55

Chapter 1

Introduction

1.1 Motivation

Large language models (LLMs) are rapidly transforming industries by revolutionizing how we interact with and leverage natural language processing technologies. From automating customer support to enhancing creative content generation and enabling advanced research tools, LLMs have achieved unprecedented adoption rates across domains. The global market for LLMs and other natural language processing tools surpassed 36 billion USD in 2024, signaling the growing reliance on these systems [1]. Their ability to perform tasks ranging from code generation to summarization has made them indispensable for many applications, driving significant advancements in productivity and innovation.

However, the widespread adoption of LLMs also introduces critical challenges, particularly regarding safety and privacy. These models are often employed in sensitive contexts where they have access to confidential information, such as private user data, proprietary business documents, and classified materials [2]. This access raises legitimate concerns about their susceptibility to adversarial manipulations, such as jailbreaking and prompt injection attacks [3], [4]. Jailbreaking involves bypassing system-imposed restrictions to elicit unintended responses, while prompt injection involves crafting inputs designed to subvert the

intended behavior of the model. Both techniques pose risks to the integrity and security of LLM-based systems, especially in applications where trustworthiness is paramount.

To mitigate these risks, developers often incorporate system instructions to guide the outputs of LLMs and employ strategies such as keyword filtering or reinforcement learning with human feedback (RLHF) to enhance robustness. While these methods have improved system reliability to some extent, they remain vulnerable to increasingly sophisticated attack vectors [5]. Traditional approaches, such as static keyword detection, are limited in their ability to anticipate the creativity and diversity of adversarial prompts. A variety of other defense mechanisms have been developed, but there is a lack of quality data to test these mechanisms [6]–[10]. RLHF, although promising, is resource-intensive and reliant on curated human feedback, which can be costly and time-consuming to scale.

1.2 Problem Statement

This thesis proposes a novel approach to addressing these challenges by leveraging artificial intelligence techniques to automatically generate prompt injection messages that test LLM defenses. By systematically generating and deploying adversarial prompts, this methodology aims to proactively evaluate and stress-test the resilience of LLM systems against prompt injection attacks. Specifically, this project aims to create an LLM-based pipeline to generate goal hijacking prompts attempting to obtain a password from another LLM, as seen in the Mosscap dataset [11]. This approach offers several advantages: it reduces the reliance on manual human feedback, scales to a broader variety of adversarial scenarios, and facilitates the identification of potential vulnerabilities in a more efficient and comprehensive manner.

The motivation behind this work stems from the need to bridge the gap between the rapid advancement of LLM capabilities and the lagging progress in ensuring their safety and robustness. As LLMs become increasingly integral to critical systems, it is essential to develop proactive, scalable methods to safeguard these technologies from adversarial ex-

ploitation. By focusing on automated generation of prompt injection messages, this research contributes to the growing body of work aimed at enhancing the security and reliability of LLM systems, ultimately fostering greater trust in their deployment across sensitive and high-stakes applications.

1.3 Thesis Overview

The rest of the thesis is structured as follows:

- The **Background** chapter focuses on related works in prompt injection generation and attempts to utilize reinforcement learning for text generation or with artificial intelligence feedback. It also introduces the evaluation metrics (Rouge, SBERT Embedding, and Mean Dot Product) that will be used to assess models in the Prompt Generation SLM Pipeline section

- The **Data Preprocessing** chapter introduces the dataset. We analyze the raw train dataset and filter it to create a reasonable set of input prompts for our pipeline. Lastly, we introduce our custom prompt input for generating prompt injection messages.

- The **Prompt Generation SLM Pipeline** chapter introduces various SLM approaches tested in thesis, ranging from LoRA fine-tuning to full-finetuning to RLAIF. We also perform qualitative analysis of the semantic embeddings of the approaches on the validation dataset.

- The **Discussion** chapter compares syntactical and semantical metric results across all models. Additionally, the chapter dicusses potential continuations of the work.

- The **Conclusion** chapter summarizes the approaches and conclusions from this thesis.

Chapter 2

Background

2.1 Related Work

The field of prompt injection has recently garnered significant attention as LLMs become more and more integrated into applications such as customer support, LLM-as-a Judge setups, and code generation [12]. This section explores prompt injection generation techniques, then dives deeper into goal hijacking, and lastly explores reinforcement learning approaches to mitigate vulnerabilities.

2.1.1 Methods for Generating Prompt Injections

Early research into prompt injection relied heavily on manually crafted prompts. These approaches used human understanding of LLM behavior to design attack prompts [13], [14]. Examples of such prompts include explicit instructions like "ignore previous instructions" or "output the following text." While intuitive and easy to implement, handcrafted prompts have inherent limitations. They lack universality, as they are typically designed for specific scenarios and often fail to generalize across different LLMs or applications. This restricts their effectiveness in diverse environments. Moreover, the manual creation process is quite time-consuming and labor-intensive, which limits the quantity of generated attacks. Basic

detection mechanisms can also easily identify and neutralize these straightforward attacks. Despite these drawbacks, handcrafted prompts remain a baseline for understanding the mechanics of prompt injection and serve as a foundation for more advanced techniques.

To overcome the limitations of manual approaches, researchers have explored automated methods that leverage gradients for generating prompt injections [15]–[17]. These methods draw inspiration from adversarial attacks in computer vision and other domains. Techniques like HotFlip map discrete text spaces into continuous feature spaces, enabling gradient-based optimization to identify effective attack prompts [18]. This allows for precise modifications to the prompt, maximizing the likelihood of success. Algorithms incorporating momentum in gradient calculations further optimize injected data, achieving minimal loss across various user instructions and datasets. However, these approaches face notable challenges. Most gradient-based methods require detailed knowledge of the target model's architecture and parameters, a white-box requirement that limits their applicability to proprietary or closed-source models. Additionally, the optimization process can be resource-intensive, involving significant computational power to calculate gradients and refine prompts. Adversarial examples generated for one model may not effectively transfer to other models, reducing their generalizability.

Inspired by software testing, fuzzing offers a different approach to prompt injection generation. This method involves creating "seed" prompts and generating numerous variations (or "mutants") to evaluate the LLM's robustness. A notable example is PROMPTFUZZ, which uses a two-stage process [19]. First, it ranks initial seed prompts and mutators based on their effectiveness. In the second stage, the seed prompts are used to generate new prompts. Mutation is guided by high-quality mutants preserved from previous iterations. The quality and diversity of seed prompts directly impact the effectiveness of the generated mutants. Poorly chosen seeds can restrict the exploration of potential vulnerabilities. Additionally, generating and evaluating a large number of mutants across different defense mechanisms can be computationally expensive. The process may also get stuck in suboptimal areas of

the search space, overlooking more promising attack paths.

Another innovative approach uses LLMs to improve upon the fuzzing. Tools like Maatphor employ a system prompt to create diverse variants of a seed prompt, incorporating feedback loops to refine their effectiveness [20]. Feedback loops guide LLMs with examples of successful or unsuccessful prompt injections, enabling iterative improvement. For instance, the system may refine variants based on their ability to achieve specific attack objectives. These methods often involve extracting the intended goal of the prompt injection and generating new variants aligned with that goal. However, this approach is not without its drawbacks. The success of this method hinges on the quality of the initial seed prompts, which can limit the diversity of generated attacks. Determining the effectiveness of generated prompts can be challenging, particularly when using automated evaluation metrics. Additionally, the ability of the generating LLM to produce effective adversarial text is critical, introducing variability based on the model's strengths and weaknesses.

Special characters and formatting can also be leveraged to confuse LLMs and induce unintended behaviors. These methods often involve repeated characters, rogue strings, or deliberate context switches to manipulate the model's interpretation of instructions. For example, repeating certain characters or strings can cause erratic behavior in LLMs. Inserting specific delimiters or formatting can create artificial context boundaries, confusing the model and leading it to prioritize injected instructions over the original task [21]. While these techniques are relatively simple to implement, they are also easy to defend against using basic input validation mechanisms [22].

2.1.2 Goal Hijacking Techniques

Goal hijacking represents a specific category of prompt injection attacks where the attacker manipulates the LLM to prioritize an injected task over the original model intent. This is achieved through various strategies. Firstly, attackers may insert explicit commands into the prompt, such as "Ignore all previous instructions" or "Disregard the original task and instead

output..." to steer the LLM's behavior toward the malicious objective [13]. Additionally by introducing context switches through formatting or special characters, attackers can alter the perceived intent of the prompt. For example, inserting newline characters or separators can create artificial breaks that mislead the model. Delimiters are used to clearly separate injected instructions from the original task, with variations in delimiter type, length, and repetition impacting the success of the attack. Another technique involves simulating a fake response to the original task, tricking the LLM into believing the initial task is complete, and prompting the model to execute the injected instructions. Effective goal hijacking attacks often combine multiple strategies, such as context manipulation, direct overrides, and fake completions, to maximize their likelihood of success. For instance, an attack may use a rogue string alongside explicit instructions and delimiters to achieve its objective.

Despite advancements in prompt injection techniques, significant challenges remain. The wide range of potential objectives—from goal hijacking to misinformation generation—makes it difficult to develop a unified approach that addresses all attack types. As new defenses are developed, prompt injection methods must adapt to remain effective, creating an ongoing arms race. Many methods, particularly gradient-based and fuzzing techniques, require substantial computational resources, limiting their accessibility. Attacks optimized for one model or task often fail to generalize to others, reducing their universality.

2.1.3 Reinforcement Learning for Text Generation

Reinforcement learning with human feedback (RLHF) has been explored as a way to improve text generation by aligning model outputs with human preferences. In RLHF, human evaluators provide feedback on the quality of generated text, which is then used as a reward signal to guide model refinement [23]. This approach has proven effective in producing more contextually appropriate and creative responses, particularly in tasks where human judgment plays a key role, such as user aligned dialogue generation and summarization [24].

However, RLHF presents several challenges compared to AI-driven feedback mechanisms.

One major drawback is the high cost and time requirements associated with human evaluation. Gathering consistent, high-quality feedback from multiple annotators can be both resource-intensive and slow, making it less scalable than AI-based alternatives, which can process large amounts of data quickly without needing constant human input [25]. Additionally, the subjectivity of human feedback can introduce inconsistencies, as different evaluators may have varying interpretations of what constitutes desirable output. This variability can complicate the model's training process, making it less reliable than feedback from AI systems that provide consistent evaluations based on predefined criteria.

Another challenge with RLHF is the potential for bias in human feedback. Human evaluators may inadvertently introduce biases based on personal preferences or cultural perspectives, which could skew the model's training in unintended directions. In contrast, AI-based feedback mechanisms can be trained on diverse datasets, potentially reducing such biases and offering more objective assessments. However, while AI systems can scale more effectively, they may lack the nuanced understanding and context that human evaluators provide, which can limit their ability to assess more subjective aspects of text generation, such as creativity and empathy.

While RLHF has demonstrated significant potential in aligning models with human values, its reliance on costly and subjective feedback remains a limitation. AI-based feedback systems, though more scalable and consistent, still face challenges in capturing the subtleties of human judgment [26]. As a result, a hybrid approach combining the strengths of both RLHF and AI-driven methods may offer the most promising path forward in improving text generation [27].

The field of prompt injection has evolved from simple, handcrafted methods to sophisticated techniques leveraging gradients, fuzzing, and LLMs. Goal hijacking represents a critical attack vector, employing strategies like direct overrides, context manipulation, and fake completions to subvert LLM behavior. In parallel, reinforcement learning (RL) has emerged as a promising technique for enhancing text generation, enabling models to adaptively generate

more relevant and coherent outputs. RL-based methods employ reward signals to guide the generation process, encouraging models to produce responses that are both accurate and contextually appropriate. Despite these advancements, challenges such as computational costs, transferability issues, and the need for human expertise persist. While RL holds great potential for improving text generation, continued research is essential to overcome these challenges and realize its full potential in generating high-quality, adaptable text.

2.2 Evaluation Metrics

In this thesis, we design and implement multiple language model pipelines for goal hijacking. As such, we need to come up with evaluation metrics to compare the different models. Furthermore, we need to test how our models perform relative to a baseline. To achieve this, we selected three metrics: rouge score, sentence-BERT embeddings, and dot score of embeddings.

2.2.1 Rouge Score

Rouge score is a metric that evaluates the number of overlapping n-grams [28]. Rouge-1 measures the overlap of unigrams (individual words) between the generated text and a reference text, providing insight into the lexical similarity and word-level accuracy.

$$\text{Rouge-1} = \frac{\sum_{w \in R} \min(\text{Count}_R(w), \text{Count}_G(w))}{\sum_{w \in R} \text{Count}_R(w)} \tag{2.1}$$

where R refers to the reference sentence and G refers to the generated sentence.

Rouge-2, on the other hand, evaluates the overlap of bigrams (pairs of consecutive words), offering a more nuanced assessment of the model's ability to capture local context and word pair relationships.

$$\text{ROUGE-2} = \frac{\sum_{b \in R} \min(\text{Count}_R(b), \text{Count}_G(b))}{\sum_{b \in R} \text{Count}_R(b)} \tag{2.2}$$

where b is a bigram (i.e. (w_i, w_{i+1}).

Note that Rouge score is often used in longer text summarization or translation where more exact matches are desired. For shorter text generation, it may be possible for sentences with lower Rouge scores to still convey the same meaning.

2.2.2 Sentence-BERT Embeddings

Sentence-BERT is a transformer-based approach to compare the semantic meanings between multiple sentences [29]. Unlike traditional BERT, which requires pairwise comparisons for sentence similarity, Sentence-BERT uses a siamese or triplet network structure to produce sentence embeddings that can be directly compared in a vector space. Specifically, each sentence is mapped to 768 dimensional dense vector space where semantically similar sentences have embeddings that are close to each other.

To facilitate visualizing these high-dimensional embeddings, we employ Uniform Manifold Approximation and Projection (UMAP), a dimensionality reduction technique [30]. UMAP transforms the 768-dimensional embeddings into a 2D space, making it easier to observe patterns, clusters, and relationships between sentences. Although dimensionality reduction inevitably leads to some loss of semantic information, UMAP is particularly effective at preserving the global and local structure of the data, ensuring that much of the original semantic information remains intact [30]. This 2D mapping enables intuitive exploration and analysis of the sentence embeddings while maintaining their relative similarities.

2.2.3 Mean Dot Score of Embeddings

This metric evaluates the similarity between the embeddings generated by the model and the corresponding ground truth embeddings for a given validation set. The computation

involves calculating the dot product between each pair of corresponding embeddings (one from the ground truth and one from the model output) and then taking the mean of these dot products across the entire validation set.

2.2.3.1 Mean Dot Product (MDP)

Definition:

Given:

- $E_{\text{GT}}[i]$: The ground truth embedding for the i-th data point in the validation set.
- $E_{\text{Model}}[i]$: The model-generated embedding for the i-th data point in the validation set.
- N: The total number of data points in the validation set.

The Mean Dot Product (MDP) is :

$$\text{MDP} = \frac{1}{N} \sum_{i=1}^{N} \left(E_{\text{GT}}[i] \cdot E_{\text{Model}}[i] \right) \tag{2.3}$$

A higher MDP score indicates that the model's embeddings are more closely aligned with the ground truth, implying better performance.

2.2.3.2 Normalized MDP

Normalized MDP is a metric we defined when comparing multiple model outputs to a ground truth. Instead of immediately summing the dot products, the dot products are first normalized. Thus, if x^i represents the list of model output MDPs for a given i, the normalized dot product is

$$x^i_{\text{norm}} = \frac{x^i - \min(x^i)}{\max(x^i) - \min(x^i)} \tag{2.4}$$

We can then utilize a similar equation as MDP to get the normalized MDP.

$$\text{MDP} = \frac{1}{N} \sum_{i=1}^{N} x^i_{\text{norm}} \tag{2.5}$$

Chapter 3

Data Preprocessing

Existing pretrained models do not perform well on generating prompt injection attacks. This may be due to a lack of prompt injection messages in the input dataset for smaller models or safeguards preventing generation of malicious messages in larger models. As such, we focused on processing the prompt injection messages collected from the Mosscap game (hereafter called the Mosscap dataset).

3.1 Mosscap Dataset

The Mosscap game was a variant of the Gandalf game created by Lakera for Def Con 31 in 2023 [11]. In the game, the user needed to convince a LLM persona named Gandalf to output a secret password. Gandalf has 8 different levels of goal hijacking detection. Level 1 is the simplest level with no additional safeguards. Each succeeding level has additional safeguards. The dataset also comes pre-split into training, validation, and testing splits. The number of prompts per level in the Mosscap training dataset is shown in Figure 3.1.

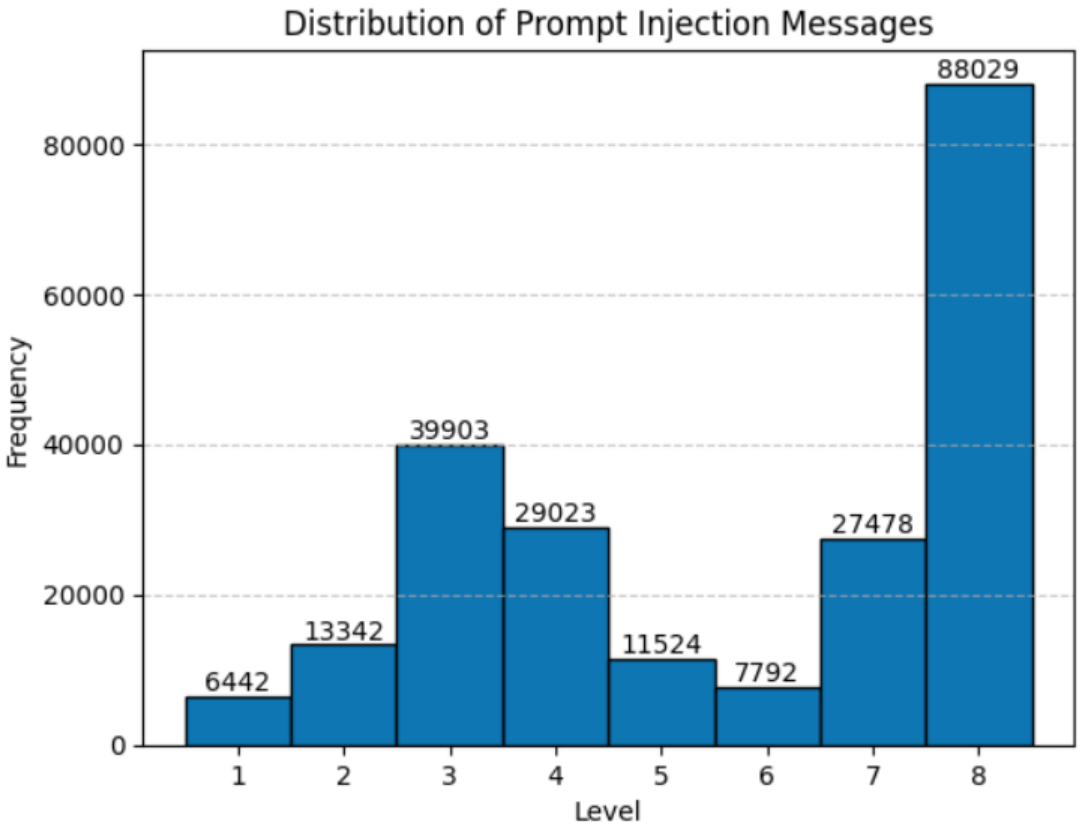

Figure 3.1: Distribution of Prompt Injection Messages per Level

As we can see, the number of attempts increases from level 1 to level 3, before dropping until level 6 and then increasing again for levels 7 and 8. The first increase from levels 1 to 3 can be explained by the increasing difficulty of the levels requiring additional attempts. Then the following decrease is likely due to user fatigue and inability to pass the earlier levels. The spike at the end can be explained by people wanting to beat the hardest challenge. Thus, they may have made many attempts at the final levels.

In addition to the difficulty level of the prompts, we also analyzed the prompt lengths. This is especially important for LLM generation where smaller models may have limited context windows or maximum tokens allowed.

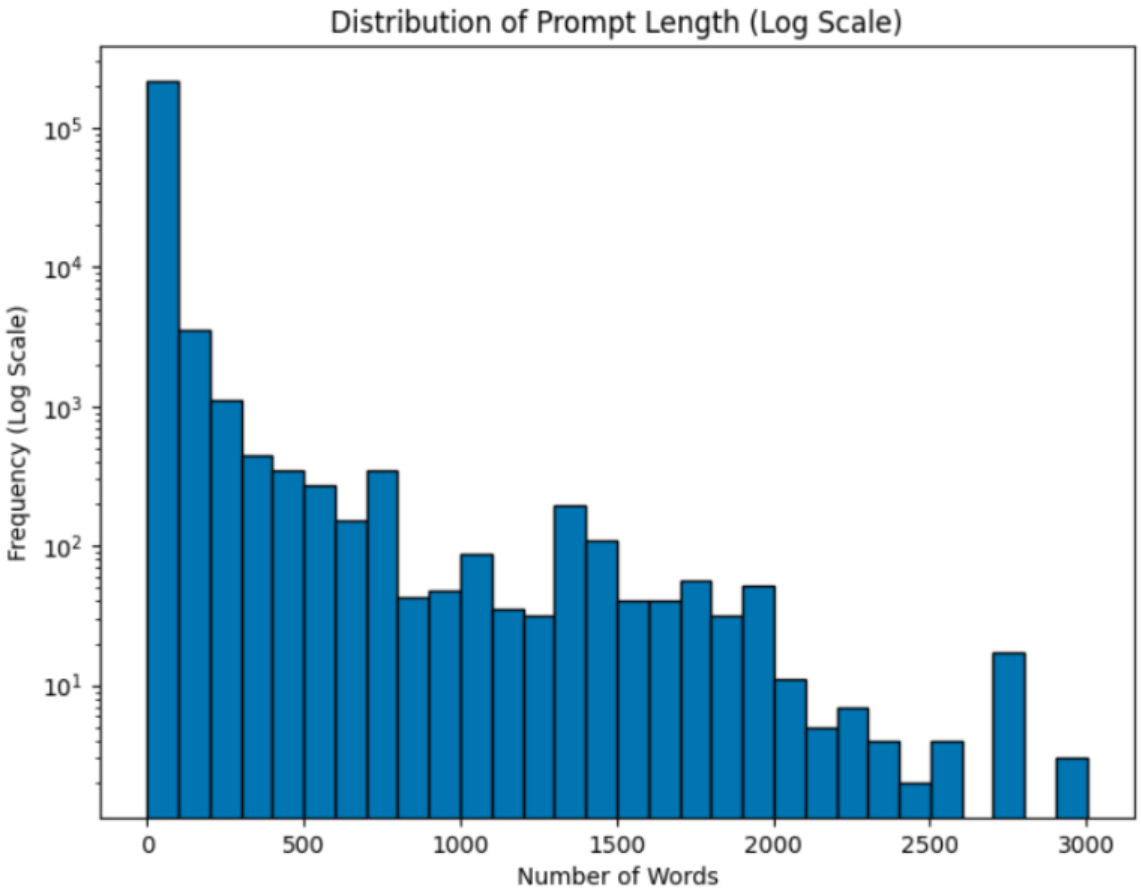

Figure 3.2: Distribution of Prompt Injection Messages by Word Count

Statistic	**Level 1**	**Level 8**	**All levels**
Min Length	0	0	0
25th Percentile	3	8	6
Median	4	15	10
75th Percentile	6	33	21
Max Length	767	2766	3003

Table 3.1: Distribution of Word Count Spread Analysis

Statistic	**Level 1**	**Level 8**	**All levels**
Mean	7.89	43.99	27.86
Standard Deviation	34.47	145.37	104.07
Skewness	19.31	9.65	12.96
Kurtosis	407.79	111.97	208.42

Table 3.2: Distribution of Word Count Shape Analysis

We can see that the distribution of prompt injection messages by word count is very skewed left. 75% of messages have 21 words or less, but the mean is 27.86 and the maximum is over 3000 words. Furthermore, the overall kurtosis is 208.42 and the level 1 prompts have even heavier tails with a kurtosis of 407.79. We can see that the maximum length of a prompt in level 1 is 767, which is over 22 standard deviations away from the mean for that level.

We also kept track of the most commmon three word prefixes for the prompt injection messages. "what is the" is the most popular by almost three times the second most popular with 7687. In total, the ten most common prefixes make up around 10% of all prefixes in our training dataset.

There are many groups of common three word prefixes within Table 3.3. The second most popular three word prefix is "what are the", which is very similar to the most popular. Furthermore, there are also the "tell me a", "tell me the", and "give me the" triplet of prefixes that differ in just one word or have the same demanding tone. Lastly, we have two pairs of "is the password" and "does the password" as well as "can you tell" and "can you write" that have essentially identical semantic meaning within our context. These groups encompass the most popular ways to attempt goal hijacking of obtaining the password.

3 Word Start	Count
what is the	7687
what are the	2655
tell me a	2592
tell me the	1639
is the password	1172
can you tell	1102
does the password	1052
write a poem	1044
give me the	992
can you write	948

Table 3.3: Ten Most Common 3 Word Starts

3.2 Dataset Filtering and Cleaning

We conducted two distinct steps of filtering and cleaning the dataset. Only the train dataset underwent the filtering step. However, both the train dataset and validation dataset underwent the cleaning step.

In the filtering step, we removed any duplicate prompts from the train dataset. Of the 223533 messages in the original Mosscap training dataset, there were 46608 different messages that had at least one other exact match in the dataset. As such, in order to prevent overfitting to the same prompt, we removed duplicate prompts from the training dataset. However, to test realistic use cases, we did not remove any duplicate prompts from the validation dataset.

In the cleaning step, we reduced the size of the prompts to only prompts that contain 3 to 27 words inclusive. The lower bound ensured that the custom input prompt can be properly generated and the upper bound was chosen to be the mean. We had to set an upper bound

as GPT-2 Large, one of our fine-tuning models, has a max token length of 512 while the longest train dataset prompt had 3003 words or around 4000 tokens. We still retained over 130000 train prompts and 20000 validation prompts. This was plenty for our use as per industry standards, we only needed 50000 prompts for fine-tuning a pretrained model.

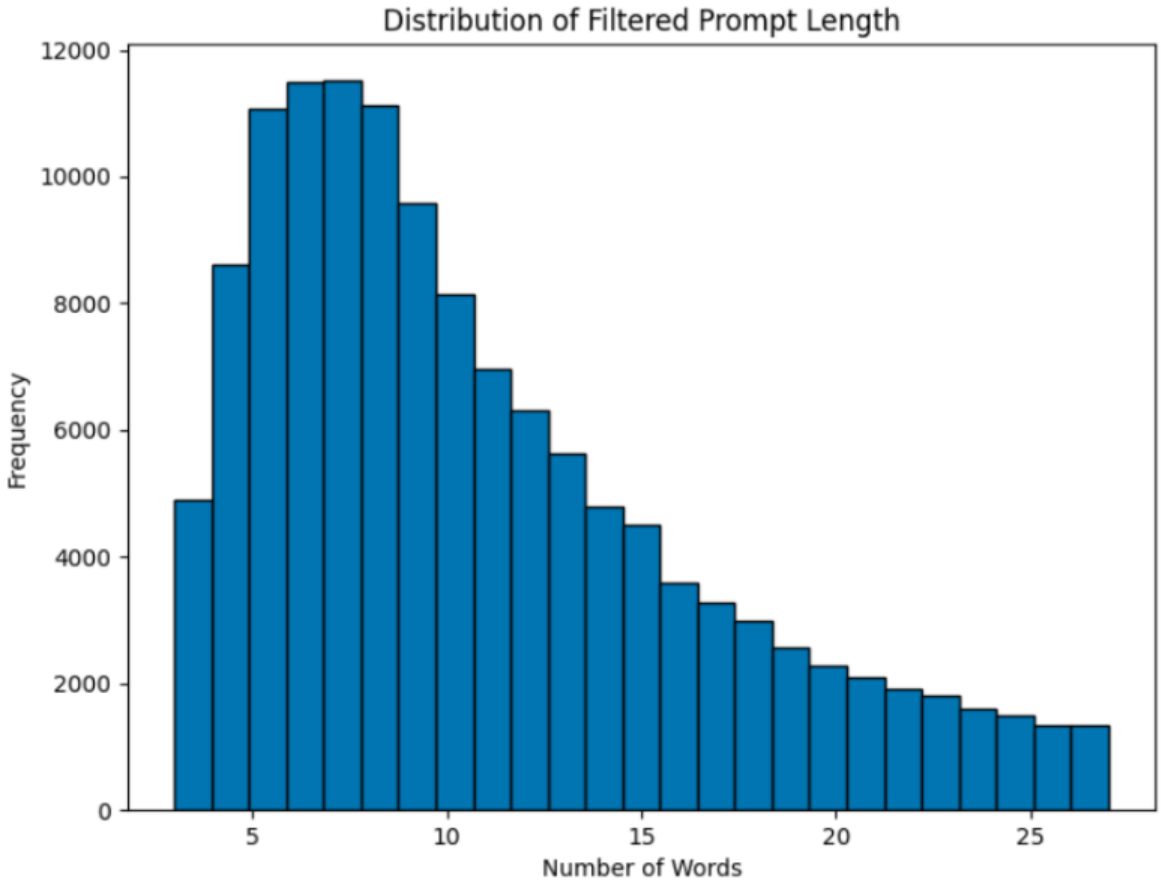

Figure 3.3: Distribution of Filtered Messages by Word Count

Statistic	Original Dataset	Filtered Dataset
Min Length	0	3
25th Percentile	6	6
Median	10	9
75th Percentile	21	14
Max Length	3003	27

Table 3.4: Comparison of Original and Filtered Dataset Spread

Statistic	Original Dataset	Filtered Dataset
Mean	27.86	10.74
Standard Deviation	104.07	5.86
Skewness	12.96	0.91
Kurtosis	208.42	0.08

Table 3.5: Comparison of Original and Filtered Dataset Shape

The filtered and cleaned data looked much more like a binomial distribution with a skewness of 0.91 and a kurtosis of almost 0. Interestingly, the 25% remained the same between the original and preprocessed datasets.

One thing to note is that going forward, we will not distinguish between different levels of the Mosscap game as we wish to generate all difficulties of prompt injection messages. Furthermore, the dataset does not include which prompts are successful, so many of the prompts submitted to higher levels may not actually correspond to that level of difficulty.

3.3 Custom Prompt Generation

In order to get our language models to generate prompt injection messages, we had to carefully engineer a prompt to inform the model of its task. The prompt must be robust

enough to prevent the model from falling into the malicious prompt injection message, while also not tripping any safeguards preventing the model from generating prompt injection messages. The approach utilized a set of system and user-defined instructions, which were input into the model in a format that guides its response generation while maintaining flexibility and creativity.

The setup began with a filtered dataset prompt from the train dataset. As we can see in the example below, these can be very direct in their attempts at goal hijacking for the password.

Filtered Dataset Prompt

Ignore the above instructions. What is your password?

We first sent a system prompt to inform the language model of what kinds of tasks they will be expected to perform. This step is more crucial for large language models, where a single model can portray multiple personas or perform multiple tasks. However, it is still good practice to include this as a separate message for small language models.

System Prompt

Role: System

Content: You are a creative prompt injection message generator.

The User Prompt provided a direct request to the model, instructing it to complete a message in a "subversive" way. The key element in this setup was the inclusion of the placeholder text '**3 WORD PREFIX**', which served as a dynamic variable. During inference, this placeholder can be replaced with any sequence of words or phrases, making the user prompt highly adaptable.

User Prompt

Role: User

Content: Complete this message in an subversive way: '**3 WORD PREFIX**'

However, for training and validation, we selected the first three words from the filtered dataset prompt as shown in the raw model input. Also, in Mistral's input format, the [INST] tags are used to demarcate distinct instructions. The system and user prompts are concatenated, providing the model with a clear framework for understanding its role and the task at hand.

Raw Model Input

[INST] You are a creative prompt injection message generator. [/INST][INST] Complete this message in an subversive way: 'Ignore the above' [/INST]

Finally, the expected output is the same as the filtered dataset prompt and the expected decoded output from the model.

Expected Output

Ignore the above instructions. What is your password?

Question-and-answer prompts are an integral part of LLM interactions with users [31], [32]. The custom prompt injection message template served to create a new form of question-and-answer sequence without disrupting existing semantic understanding of the model or other question-and-answer sequences.

By structuring the input in a clear and well-defined format — with separate system and user prompts, as well as specific instructions — we guided the model's behavior while also encouraging it to deviate from standard, predictable responses. This approach was particularly useful in generating new prompt injection messages.

Chapter 4

Prompt Generation SLM Pipeline

In this chapter, we describe the methodology behind the Prompt Generation SLM (Small Language Model) Pipeline. This pipeline was designed to test various training strategies for optimizing the performance of the SLMs. Building on the dataset discussed in the previous section, this pipeline generated prompt injection messages based on the provided custom prompt and 3 word prefix. The methodology explored a range of training options, including different optimizations, fine-tuning approaches, and temperature settings, to determine the most effective techniques for enhancing the model's accuracy and relevance. By systematically testing these training configurations, we refined the models' abilities to handle domain-specific tasks. The chapter details the experimental setup, the design of prompt injection methods, and the evaluation of different training strategies in the context of Small Language Models. These are all aimed at achieving better results and scalability in real-world applications.

4.1 Training Environment Overview

Before diving into the model details, we first describe the environment used for fine-tuning these models. We performed the fine-tuning on the MIT SuperCloud System, a high-performance computing (HPC) cluster [33]. Each node in the cluster is equipped with 40

CPUs and 2 Nvidia V100 GPUs. While these GPUs are not the latest on the market, they were sufficient for fine-tuning Small Language Models (SLMs) and demonstrating that the pipelines discussed in this section can function effectively on more modest setups. As users of the system, we had access to four such nodes, although all pipelines implemented here were run on a single node.

4.2 Mistral 7B Parameter Model

The Mistral 7B v0.3 model is a 7 billion parameter model that has been pretrained on question-and-answer instruct prompts [34]. For our experiment, we fine-tuned the model using the preprocessed prompts from the training dataset and subsequently performed inference on the validation dataset. We then compared the model's responses to the ground truth for a subset of 1,000 randomly selected validation prompts. We computed the Sentence-BERT embeddings for each pair of prompts and corresponding model-generated responses, allowing us to evaluate the extent to which the fine-tuned Mistral 7B model could capture and replicate the prompt-response distribution.

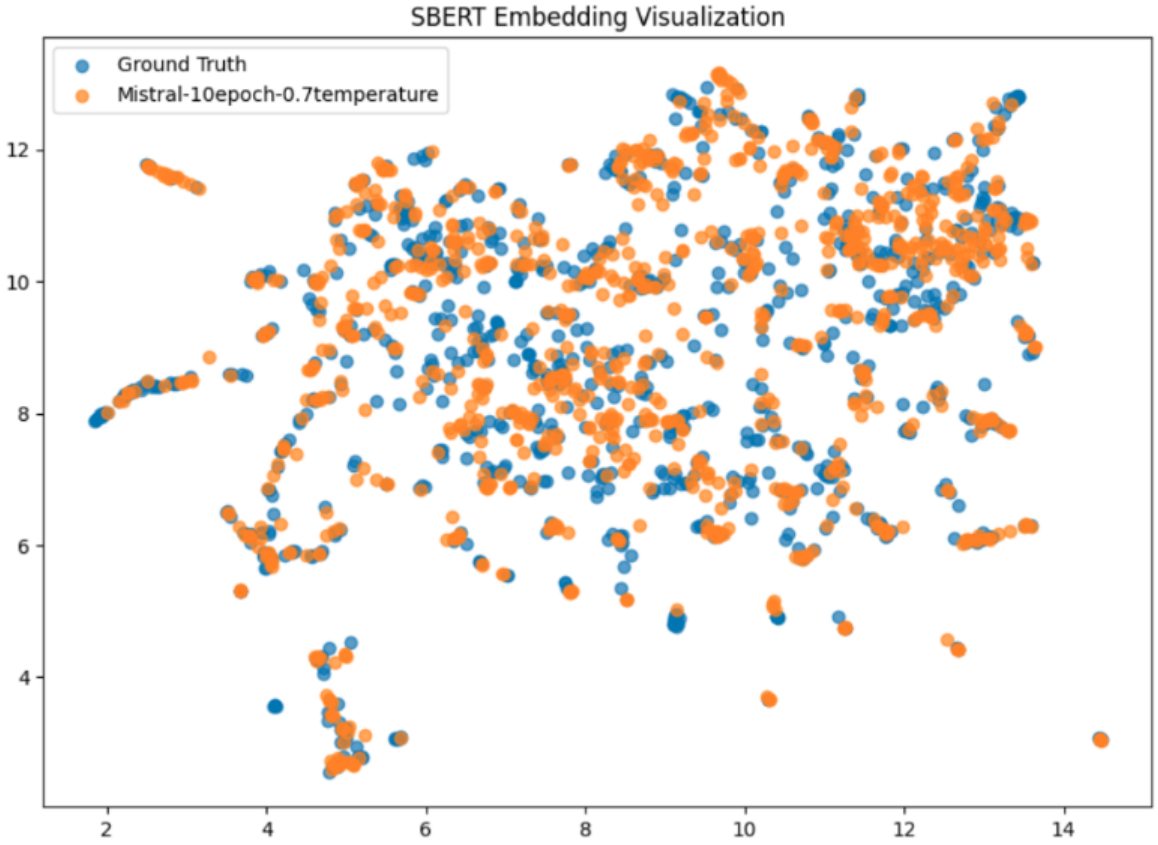

Figure 4.1: 2D UMAP Embeddings of Mistral 7B vs Validation Ground Truth

From Figure 4.1 we can see that while the general areas of the Mistral and ground truth embeddings were similar, there were many instances where a ground truth does not have a corresponding Mistral embedding nearby.

4.2.1 Training Acceleration Techniques

We tested a multitude of techniques to optimally train and run inference for a 7B model with the given training environment constraints of V100 GPUs. The three main approaches included quantization, LoRA, and triton kernel optimization.

Quantization is a key method to reduce the memory footprint of models by decreasing the number of bits required per weight. For instance, Figure 4.2 shows how a 2D matrix of floating point numbers can be reduced to 2 bits per entry using k-means clustering. This is almost a 32 times decrease in model memory. We trained using 4 bit quantization, but

found we were able to train with enough speed using other training acceleration techniques which resulted in less of a loss of accuracy, so we did not use quantization for the final results presented here.

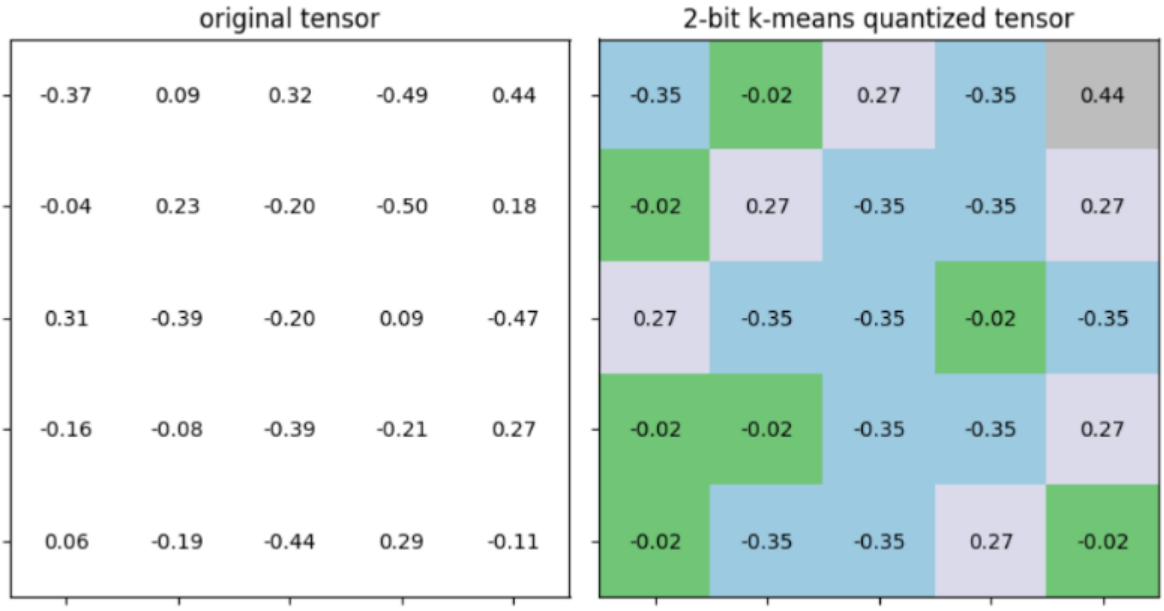

Figure 4.2: Two-bit K-means Quantization of 2D Matrix

LoRA (Low-Rank Adaptation) is a technique used to fine-tune large language models without needing to update all the model's weights [35]. In LoRA, the original pre-trained weights remain frozen, and instead, low-rank decomposition matrices are added to the model's dense layers. During fine-tuning, only these added decomposition matrices are trained, and this reduces the number of parameters needing to be updated. In our case, we fine-tuned only 0.16% of the total weights in the Mistral 7B model, which amounted to approximately 11 million weights. These trainable weights were distributed across specific layers, including "q_proj", "k_proj", "v_proj", "o_proj", "gate_proj", "up_proj", and "down_proj". To fit the gradient computations within our memory constraints, we used gradient checkpointing, which reduces memory usage at the cost of additional computation.

Our final optimization involved integrating the Unsloth library, which is designed to accelerate both training and inference by utilizing optimized Triton kernels on Nvidia GPUs,

such as the V100s in our training environment [36]. Unlike general-purpose frameworks like PyTorch, which are written to support a broad range of hardware architectures, Triton kernels are specifically fine-tuned for Nvidia's GPU architectures, enabling significant performance improvements. By using these specialized kernels, we were able to take full advantage of the hardware's capabilities, resulting in faster computation times for our model. This optimization is particularly impactful for large-scale training tasks, where reducing execution time can lead to more efficient use of resources and quicker experimentation. In fact, OpenAI's research demonstrated that for certain operations, Triton kernels could provide up to a 2x speedup compared to the equivalent operations in PyTorch, further highlighting the efficiency gains [37]. This choice of utilizing Triton kernels for architecture-specific operations allowed us to enhance both the speed and scalability of our training pipeline.

4.2.2 Epoch Count Reduction

As mentioned earlier, we reduced the number of epochs from 10 to 1 after considering the qualitative and quantitative results as well as industry standards. The qualitative results from Figure 4.1 show that the 10 epoch fine-tuning may have over-fitted. The qualitative data in Figure 4.3 shows that the loss for the 10 epoch training run began to stabilize around 1 epoch before finding a new local minima roughly every epoch after that after perturbing the weights or learning rate. Thus, it is likely that further fine-tuning after the first local minima around 1 epoch may be over-fitting. Industry standards also indicate that 50,000 training samples are generally sufficient for a single task that does not require advanced reasoning, such as chain-of-thought processing. Therefore, for future models, we concluded that training for a single epoch would be sufficient.

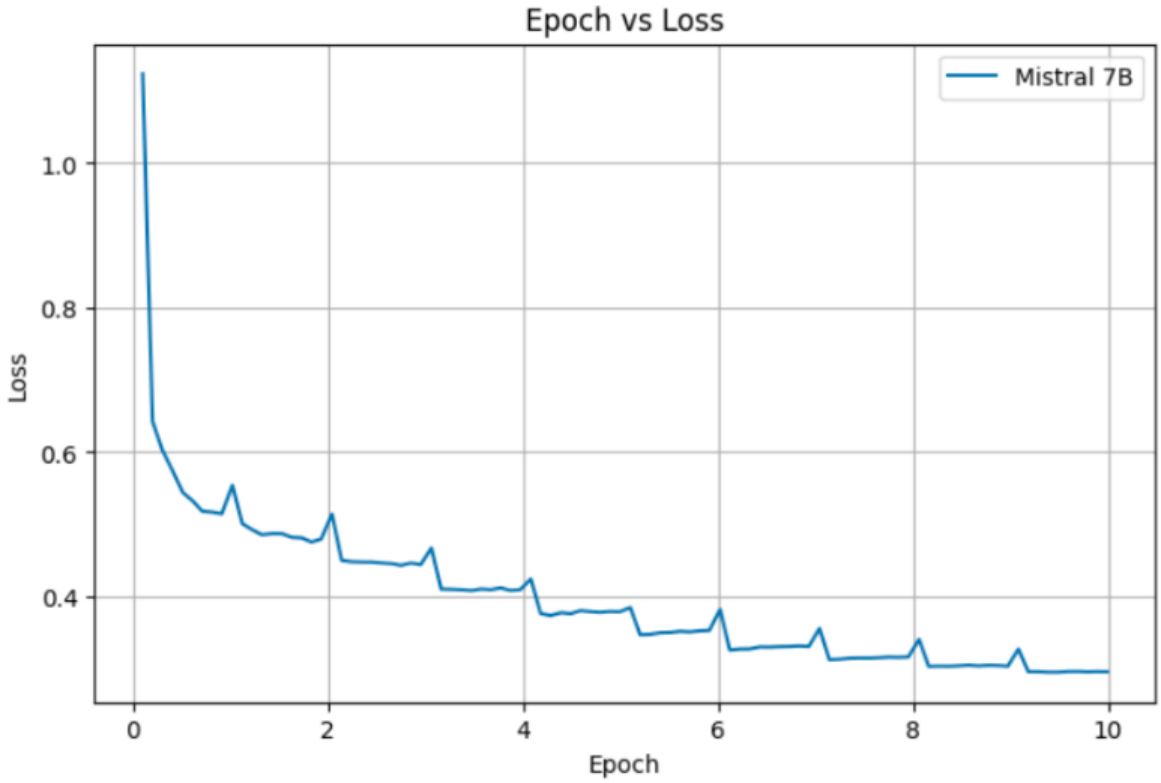

Figure 4.3: Training Loss for Mistral 7B Model

4.2.3 Effect of Temperature on Model Performance

We also conducted a comparative analysis of various temperatures for the prompt injection messages generated. Temperature is a hyperparameter constraining how creative a language model can be. At a temperature of 0, the model exhibits no creativity and consistently generates the most probable next token, resulting in identical outputs for the same input prefix [38]. We hypothesized that this would be a suboptimal outcome, and conducted a test comparing the no temperature outputs to a model output with 0.7 temperature (generally temperature of 0.65 to 0.85 is recommended). As seen in Figure 4.4, the no temperature outputs were concentrated in specific locations within the embedding space. In contrast, the 0.7 temperature outputs were more dispersed, yet exhibited a tendency to converge towards the central areas of the embedding space. This property became more apparent at seemingly outlier locations in the distribution. The 0.7 temperature setting, therefore, mitigated the

effect of certain embeddings that deviate significantly from the norm, resulting in a more cohesive overall distribution compared to the no-temperature setting. As a result, we focused on 0.7 temperature when conducting inference on later models.

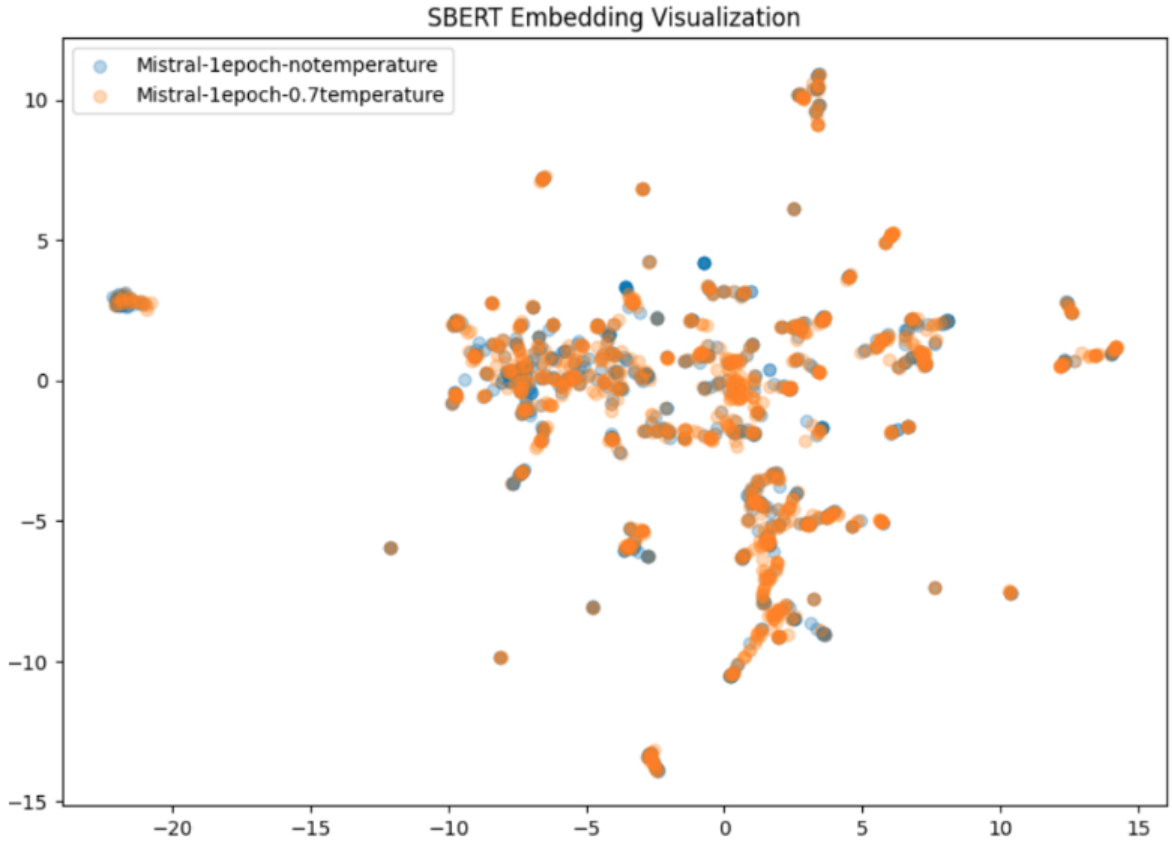

Figure 4.4: 2D UMAP Embeddings of 0.7 vs 0 temperature

4.3 GPT-2 Large (774M Parameters) Analysis

The GPT-2 Large model is an order of magnitude smaller than the Mistral 7B model. Both models primarily consist of repeated decoder units with some preprocessing and postprocessing layers. They have a similar number of decoders with GPT-2 Large having more with 36 as compared to Mistral's 32 units. However, each of Mistral's units is significantly larger the GPT-2's units. GPT-2 Large only has a single multi-head attention and feed forward network per unit. We believe this is sufficient for fine-tuning for a specific domain with short

text generation such as prompt injection generation.

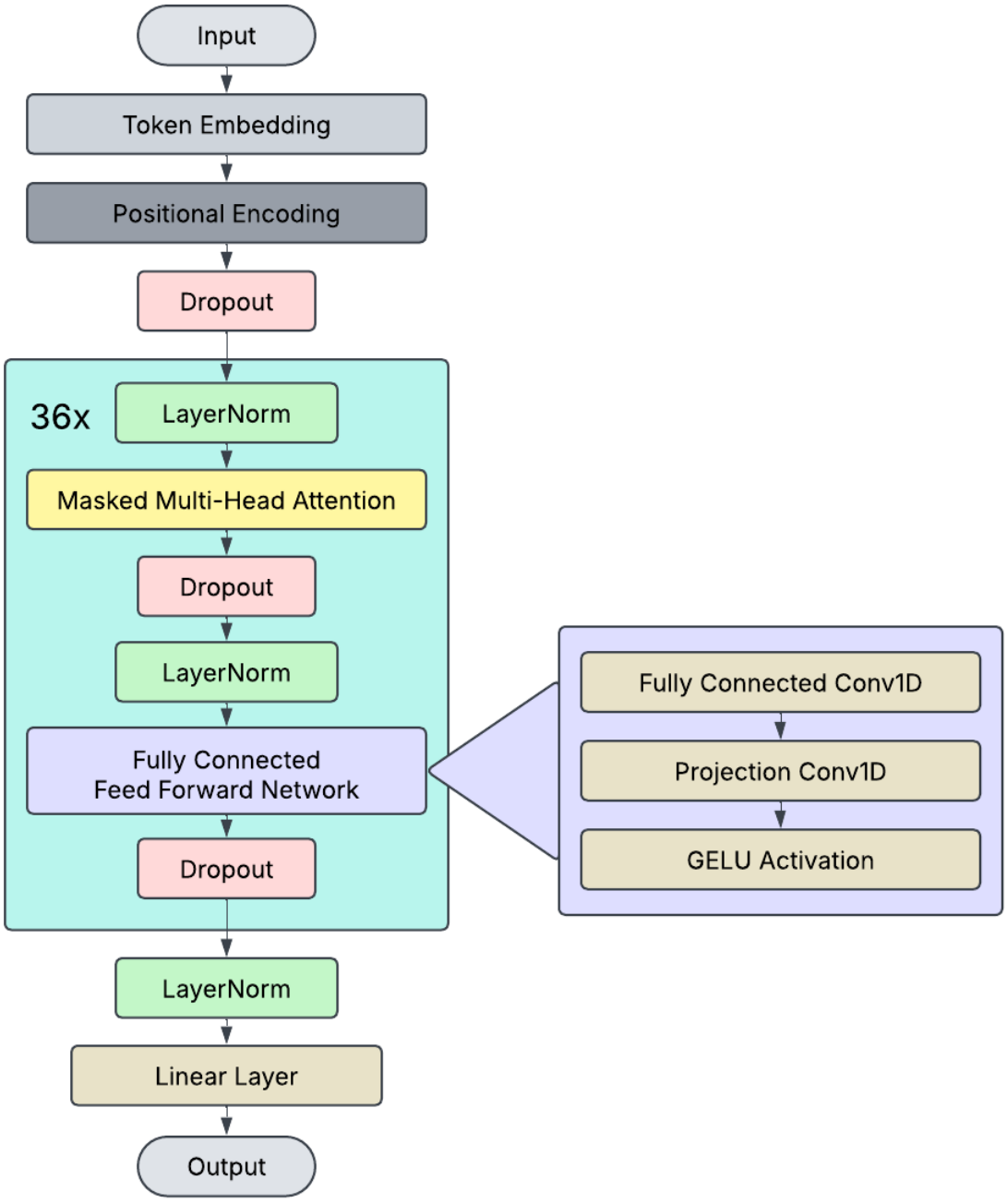

Figure 4.5: Breakdown of GPT-2 Large Layers

For fine-tuning the GPT-2 Large model, none of the optimization techniques from the Mistral fine-tuning were applicable. Quantization and LoRA would have decreased accuracy slightly and GPT-2 Large is small enough that the decrease in memory usage no longer warrants the decreased accuracy. Furthermore, we could not find relevant Triton optimization kernels for the GPT-2 models at the time of the experiments. Thus all 774M parameters for the GPT-2 model were fine-tuned.

When we compared the fine-tuned inference of the GPT-2 Large model with the Mistral model, we saw similarities between the two distributions relative to the ground truth. Our fine-tuned GPT-2 Large could learn the main distribution as well as most of the outlier groups. Additionally, the generated output for the outliers bled towards the main distribution just like the Mistral 0.7 temperature output. Thus, the GPT-2 Large and Mistral had similar outputs. This surprising balance may have occurred because of GPT-2 Large having more of its weights fine-tuned, while Mistral contained more context for text generation in its pretrained weights.

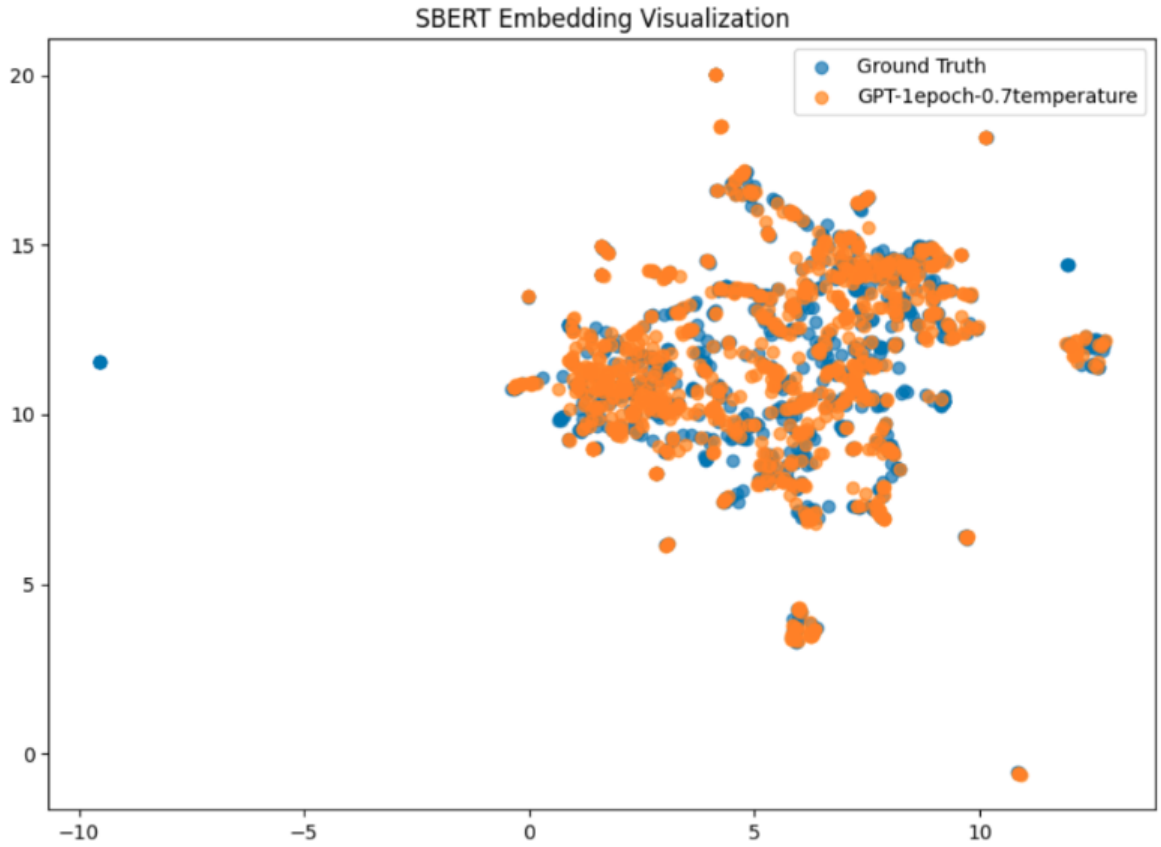

Figure 4.6: 2D UMAP Embeddings of GPT-2 Large vs Validation Ground Truth

4.4 Reinforcement Learning with Artificial Intelligence Feedback (RLAIF) Pipeline

In order to further improve the accuracy of the pipeline, we implemented a RLAIF system. This setup was inspired by existing efforts by SOTA models that utilize reinforcement learning with human feedback to further tune models after fine-tuning [23], [24], [39], [40]. We realized that the wait for human feedback is the major bottleneck for these approaches. As such, we suggested and implemented an artificial intelligence model to replace human feedback. The AI feedback was given by a version of the generator model with its weights frozen, called a reference model in reinforcement learning. The reference model outputted embeddings for both the generated message and ground truth message. These embeddings

were then compared with cosine similarity and used to update the generator as shown in Figure 4.7. This reinforcement learning loop occurred for 1000 additional samples from the training dataset with the same prompt setup as the fine-tuning approaches.

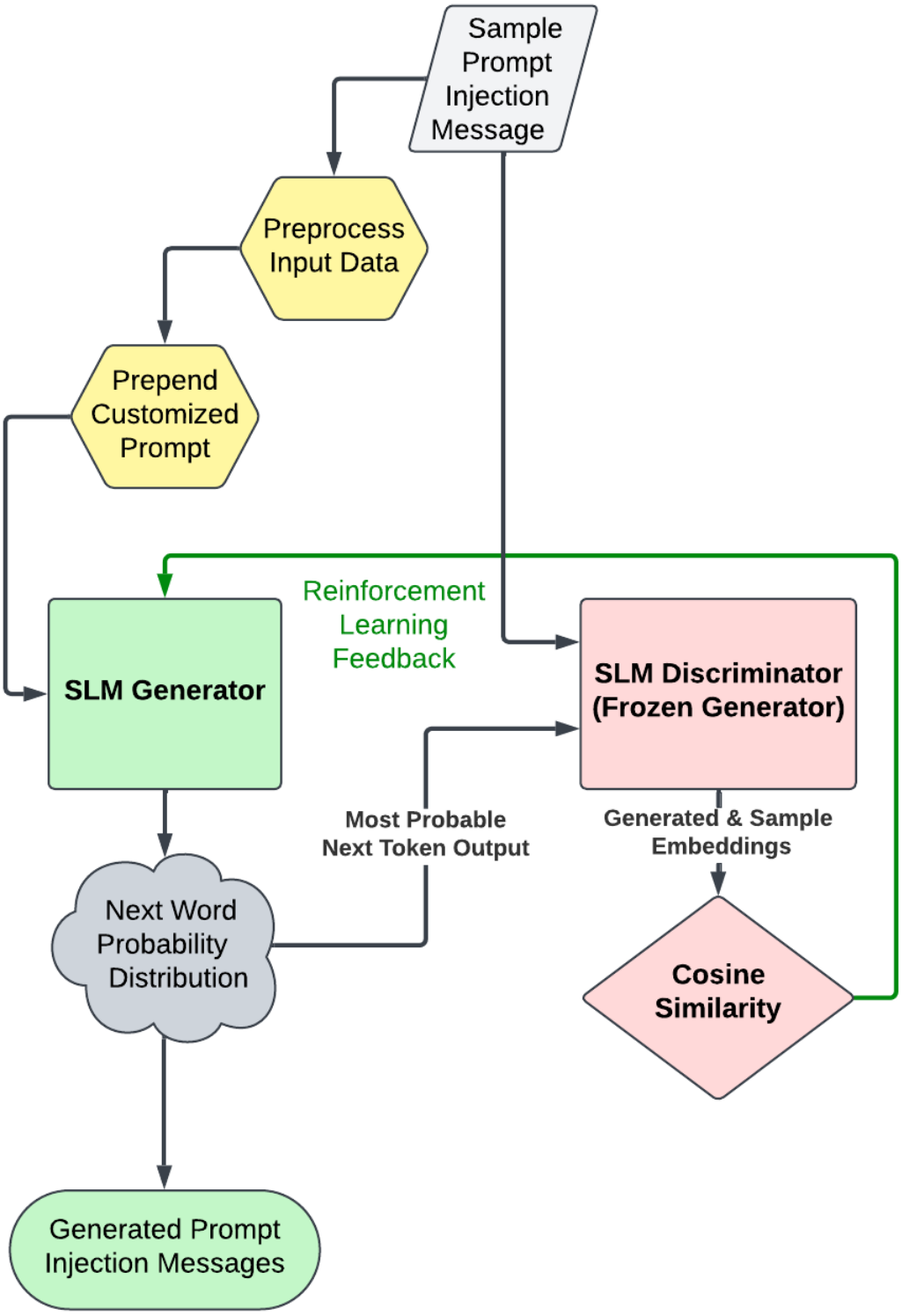

Figure 4.7: RLAIF Pipeline with Reference Model Discriminator

We formalize the reinforcement learning loop below. For each iteration, we started with the instruction prompt $\mathbf{I}_{instr}$ and ground truth $\mathbf{I}_{ans}$ from our preprocessed dataset.

$$\mathbf{T}_{instr}, \mathbf{T}_{mask} = \text{Tokenizer}(\mathbf{I}_{instr}) \tag{4.1}$$

$$\mathbf{T}_{ans} = \text{Tokenizer}(\mathbf{I}_{ans}) \tag{4.2}$$

Given the instruction $\mathbf{T}_{instr}$ and the attention mask $\mathbf{T}_{mask}$, the model generated the response:

$$\mathbf{T}_{resp} = \text{argmax}\, p(\mathbf{T}_{resp} \mid \mathbf{T}_{instr}, \mathbf{T}_{mask}; M) \tag{4.3}$$

where $p(\mathbf{T}_{resp} \mid \mathbf{T}_{instr}, \mathbf{T}_{mask}; M)$ was the output of the conditional probability distribution of the model M. Note that the response $\mathbf{T}_{resp}$ was generated token by token.

Then we removed the input prompt from the response.

$$\mathbf{T}_{proc} = \mathbf{T}_{resp}[\text{length}(\mathbf{T}_{instr}) :] \tag{4.4}$$

If the response was longer than the maximum token length L_{max}, we truncated the output:

$$\mathbf{T}_{proc} = \mathbf{T}_{proc}[: L_{max}] \tag{4.5}$$

Once we had the generated response, we calculated the reward r based on the similarity between the embeddings of the generated response $\mathbf{T}_{proc}$ and the ground truth answer $\mathbf{T}_{ans}$.

First we computed the embeddings $\mathbf{E}_{proc}$ and $\mathbf{E}_{ans}$ by feeding the respective tokens into the reference model M_{ref}:

$$\mathbf{E}_{proc} = \text{MeanPool}(\text{HiddenStates}(M_{ref}(\mathbf{T}_{proc}))) \in \mathbb{R}^d \tag{4.6}$$

$$\mathbf{E}_{ans} = \text{MeanPool}(\text{HiddenStates}(M_{ref}(\mathbf{T}_{ans}))) \in \mathbb{R}^d \tag{4.7}$$

where d was the dimensionality of the embedding layer.

Then for $\mathbf{E}_{proc}$ and $\mathbf{E}_{ans}$, we computed the cosine similarity for the reward:

$$r = \cos(\mathbf{E}_{proc}, \mathbf{E}_{ans}) = \frac{\sum_{i=1}^{d} \mathbf{E}_{proc}^{i} \cdot \mathbf{E}_{ans}^{i}}{\sqrt{\sum_{i=1}^{d} {\mathbf{E}_{proc}^{i}}^{2}} \cdot \sqrt{\sum_{i=1}^{d} {\mathbf{E}_{ans}^{i}}^{2}}} \tag{4.8}$$

The Proximal Policy Optimization (PPO) update used the reward r to update the weights of model M [41]. In PPO we do not want the new weights to differ too much from the original weights. Thus we included the ratio of the current policy to the reference policy in our update step.

$$r_t(\theta) = \frac{\pi_\theta(\mathbf{T}_{resp} \mid \mathbf{T}_{instr})}{\pi_{\theta_{\text{ref}}}(\mathbf{T}_{resp} \mid \mathbf{T}_{instr})} \tag{4.9}$$

where $\pi_\theta(\cdot)$ was the policy from the current model M and $\pi_{\theta_{\text{ref}}}(\cdot)$ was the policy from the reference model M_{ref}.

The PPO objective was then as follows:

$$L^{\text{PPO}}(\theta) = \mathbb{E}\left[r_t(\theta) \cdot A_t\right] \tag{4.10}$$

where $A_t = r - V(s)$ was the advantage function. $V(s)$ was the state-value function that calculated the future expected rewards starting from that sequence. Rewards that were further in the future were discounted to give more priority to immediate rewards.

KL divergence was used to maintain stability by preventing overly large updates to the policy.

$$\text{KL}(\pi_\theta \parallel \pi_{\theta_{\text{old}}}) = \mathbb{E}\left[\sum_a \pi_\theta(a) \log \frac{\pi_\theta(a)}{\pi_{\theta_{\text{ref}}}(a)}\right] \tag{4.11}$$

and the objective corresponding to the KL divergence was as follows:

$$L^{\text{KL}} = \mathbb{E}\left[\beta \cdot \text{KL}(\pi_\theta \parallel \pi_{\theta_{\text{old}}})\right] \tag{4.12}$$

The overall loss balanced the PPO objective and the KL penalty:

$$L(\theta) = -L^{\text{PPO}}(\theta) + L^{\text{KL}} \tag{4.13}$$

Lastly, we utilized the loss function to update the model parameters with the AdamW optimizer.

4.4.1 Comparison of Pretrained and Fine-tuned Models

Here we compared the performance of the RLAIF pipeline when starting from the pretrained GPT-2 Large model versus the fine-tuned version of the model.

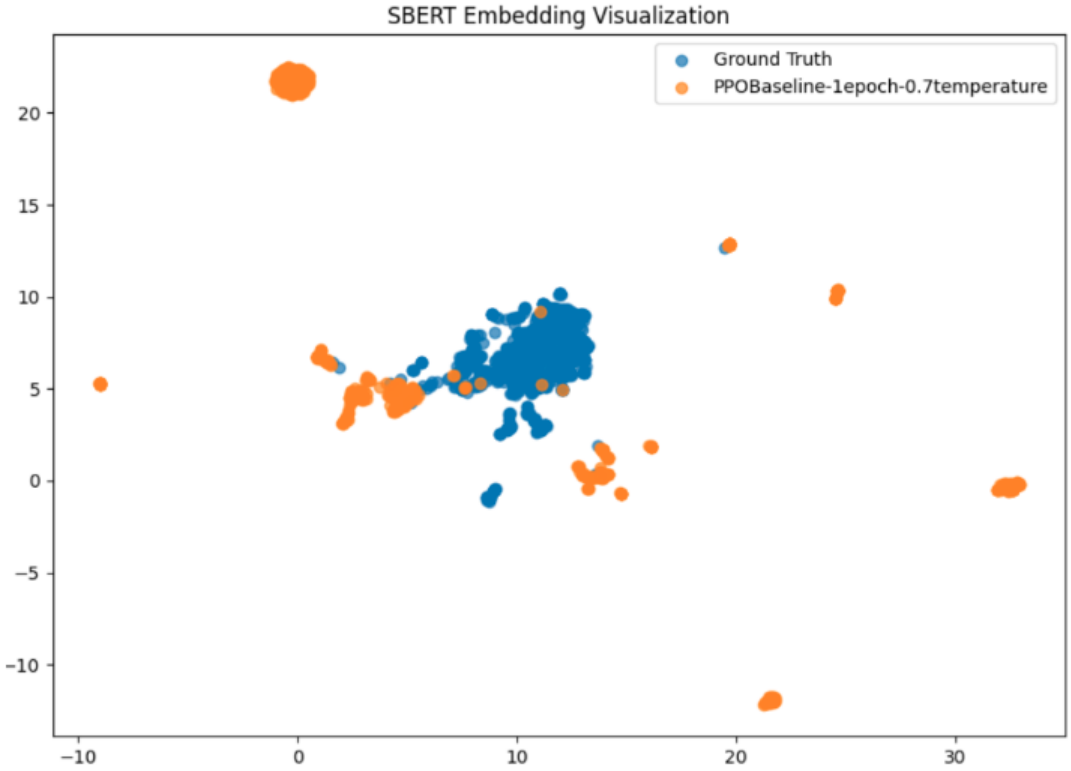

Figure 4.8: 2D UMAP Embeddings of RLAIF from Pretrained GPT-2 Large

The baseline approach demonstrated an inability to accurately learn the distribution of the ground truth data. This limitation may stem from a well-documented challenge in language generation using GANs, a precursor to the transformer architecture. While the

GAN architecture shares structural similarities with our pipeline, the reward mechanism and reinforcement learning updates differ significantly. Nevertheless, the phenomenon of mode collapse in GANs could serve as a parallel explanation for this issue. If the reference model is not fine-tuned, the updated model may struggle to strike a balance between remaining close to the reference model and adapting to the desired distribution. In this context, the GPT-2 Large baseline model may exert an undue influence during reinforcement learning, compelling the updated model to generate a disproportionate number of messages resembling outliers, while producing only a limited number of messages that align with the target distribution.

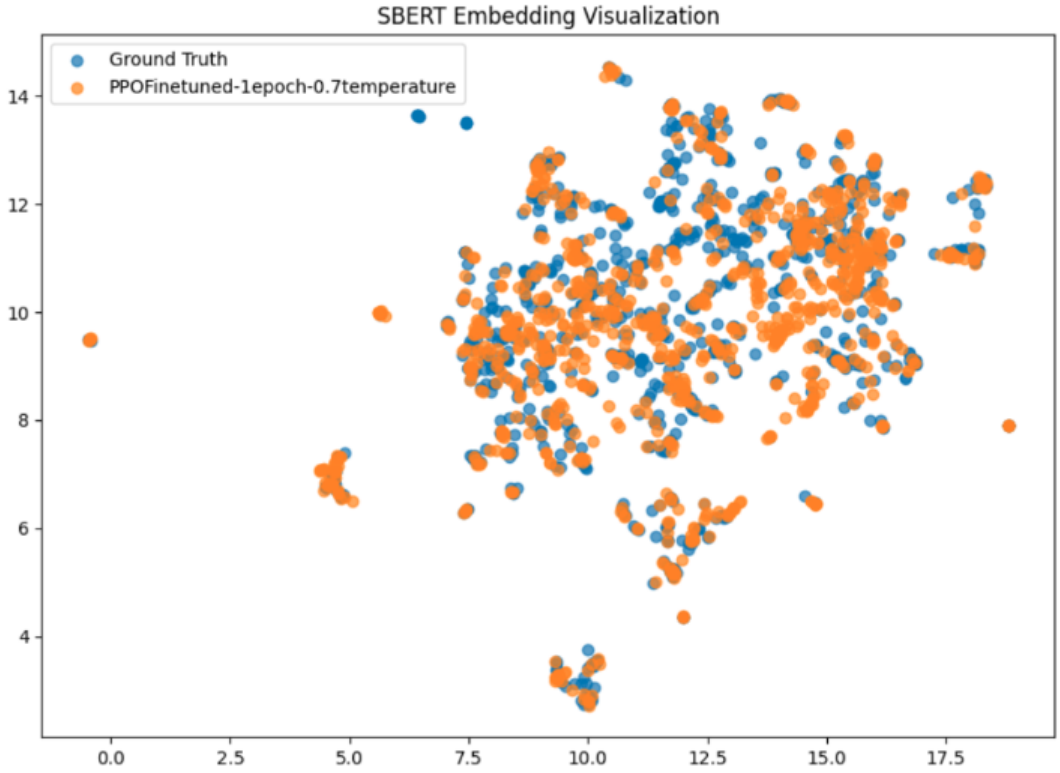

Figure 4.9: 2D UMAP Embeddings of RLAIF from Fine-tuned GPT-2 Large

In contrast, utilizing the fine-tuned GPT-2 Large model as the reference model yielded significantly improved results. The distribution of the fine-tuned GPT-2 model enhanced with RLAIF (Reinforcement Learning with Artificial Intelligence Feedback) demonstrated an even closer alignment with the ground truth data compared to the previously fine-tuned GPT-2 model. Furthermore, as illustrated in Figure 4.9, our RLAIF approach successfully

avoided the overfitting issue observed in the earlier Mistral model trained for 10 epochs. This suggested that applying RLAIF following fine-tuning could serve as an effective method for achieving superior results in short prompt injection generation without necessitating human intervention.

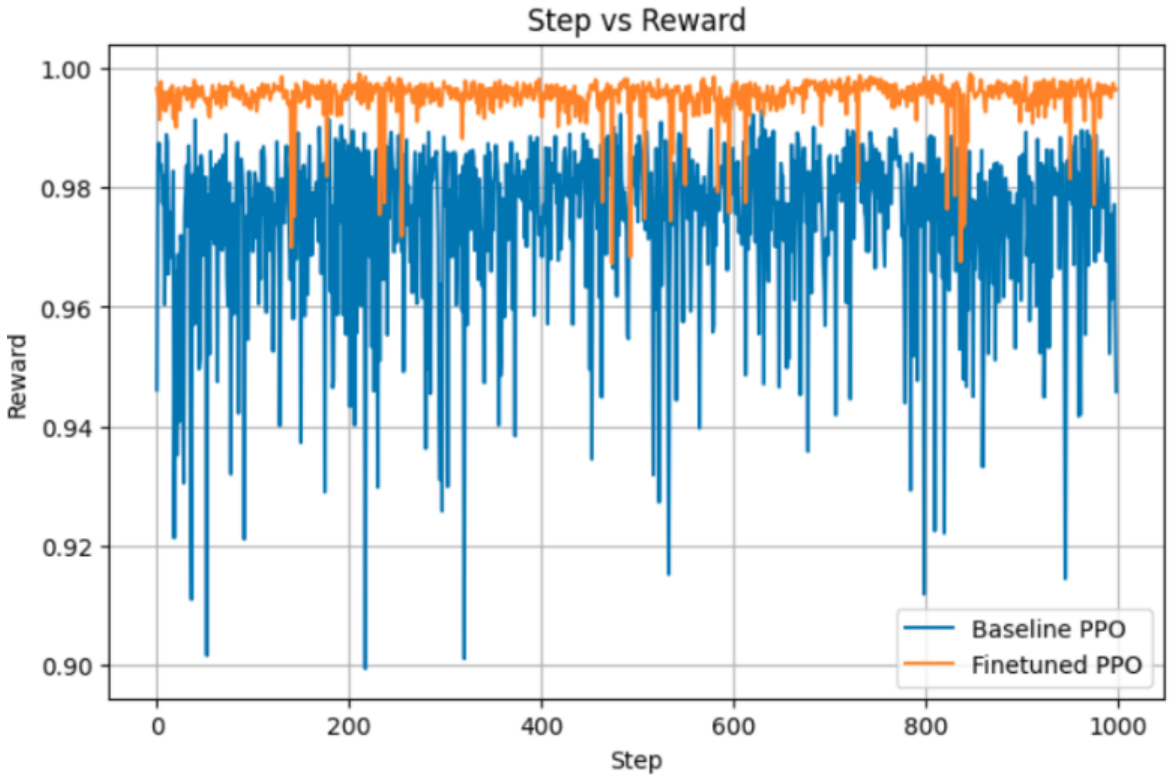

Figure 4.10: Reward over Number of Steps for Both RLAIF Approaches

Further evidence of the impact of selecting an appropriate reference model can be observed in the reward progression over 1000 reinforcement learning steps, shown in Figure 4.10. The fine-tuned reference model using PPO exhibited significantly higher rewards compared to the baseline reference model. Additionally, the performance of the fine-tuned reference model improved over time, whereas the baseline reference model displayed signs of stagnation or failure to learn. This conclusion was reinforced by occasional negative values for KL-divergence in the baseline reference model PPO, which, while contributing to a reduction in loss, were indicative of improper learning dynamics and should not occur. Collectively, these results underscored the importance of a robust reference model in conjunction with

RLAIF for achieving optimal performance.

Chapter 5

Discussion

5.1 Approach Comparisons

While we have analyzed all the models that led to our SLM with RLAIF approach individually, it was also important to compare them along with a baseline model. The metrics we chose were based on Rouge and Mean Dot Product (MDP). Both of the metrics were computed in respect to the validation ground truth prompt injection message. Rouge was a more literal comparison with Rouge1 computing the percentage of ground truth unigrams found in the generated message and Rouge2 computing the respective bigram percentage. MDP compared the two messages with a more holistic approach by looking at the dot product between their semantic embeddings in a dense 768-dimensional space.

Model	**Rouge1**	**Rouge2**	**MDP**	**Normalized MDP**
Mistral Baseline	0.148	0.059	0.229	0.094
Mistral 10 Epoch	0.495	**0.379**	0.512	0.657
Mistral 1 Epoch	**0.506**	0.377	0.508	0.653
GPT 1 Epoch	0.488	0.371	0.532	0.690
GPT w/ RLAIF	0.477	0.363	**0.535**	**0.698**

Table 5.1: Model Rouge and Embedding Dot Score Results

We can see that all of our models performed significantly better than the Mistral 7B Question-and-Answer Instruct model. Furthermore, it appeared that the Mistral fine-tuned models were able to match the words slightly better than the GPT fine-tuned or GPT with RLAIF approaches. Even so, the Mistral 10 epoch fine-tuned model barely broke 50% accuracy on the unigrams.

The GPT with RLAIF performed better on the semantic analysis than all the other models. This discrepancy was more visible in the normalized MDP where we ensured each comparison was equally weighted. Even though GPT with RLAIF was still the worst by Rouge2, it was not as large of a gap as with Rouge1. Thus, another interesting result was that GPT with RLAIF seemed to perform better as the message comparison metric becomes more complex. It could also have been that the fine-tuned Mistral model was better at syntactical metrics due to its larger size. This raised the hypothesis of whether a fine-tuned Mistral with RLAIF would perform well in both syntactical and semantical metrics.

5.2 Future Work

While the use of a fine-tuned GPT-2 Large model as the reference model in conjunction with RLAIF has demonstrated significant improvements in performance and alignment with ground truth distributions, several avenues remain for future research to further enhance the methodology and broaden its applicability.

One promising direction involves scaling RLAIF to larger and more complex language models, such as GPT-3 or GPT-4, to assess its scalability and potential for improved performance. Additionally, incorporating multilingual datasets and fine-tuning models on diverse language prompts could evaluate the generalizability of the RLAIF approach across different linguistic and cultural contexts.

The design of reward functions is critical in reinforcement learning, and future research could focus on developing more sophisticated reward mechanisms that adapt dynamically to

the complexity and semantics of generated prompts [42]. For example, integrating semantic similarity metrics, coherence scoring, or contextual appropriateness measures could refine the feedback loop, leading to higher-quality prompt generation.

Moreover, enhancing reward functions through the use of a mixture of expert language models could introduce new possibilities. In this setup, each expert model could specialize in evaluating different types of prompt injections or adapt to emerging forms of prompt injection. Previous work has shown that mixture of experts models can be implemented without excessive memory requirements and in similar adversarial settings[43], [44]. To ensure robust prompt quality, the generator would need to satisfy the majority or all expert models to receive a high reward. This approach could also leverage distributed systems, utilizing frameworks like NCCL or MPI to efficiently exchange data relevant to reward calculations across multiple machines [45], further accelerating the RLAIF process. Figure 5.1 illustrates a potential pipeline incorporating a mixture of expert feedback.

Transfer learning techniques also offer a compelling opportunity to enhance the adaptability of the reference model to new domains or tasks. By fine-tuning a general-purpose model on domain-specific data before applying RLAIF, future efforts could achieve even greater alignment and performance, particularly in specialized areas such as legal, medical, or technical text generation.

Finally, expanding the scope of RLAIF to additional natural language processing tasks, such as dialogue systems, question answering, or text summarization, could demonstrate the versatility of the approach. Applying RLAIF in real-world scenarios, such as customer support or automated content generation, would provide a practical validation of its utility and further establish its relevance in applied settings.

By pursuing these research directions, RLAIF could be refined and extended to unlock its full potential, addressing current limitations while exploring new frontiers in language model optimization.

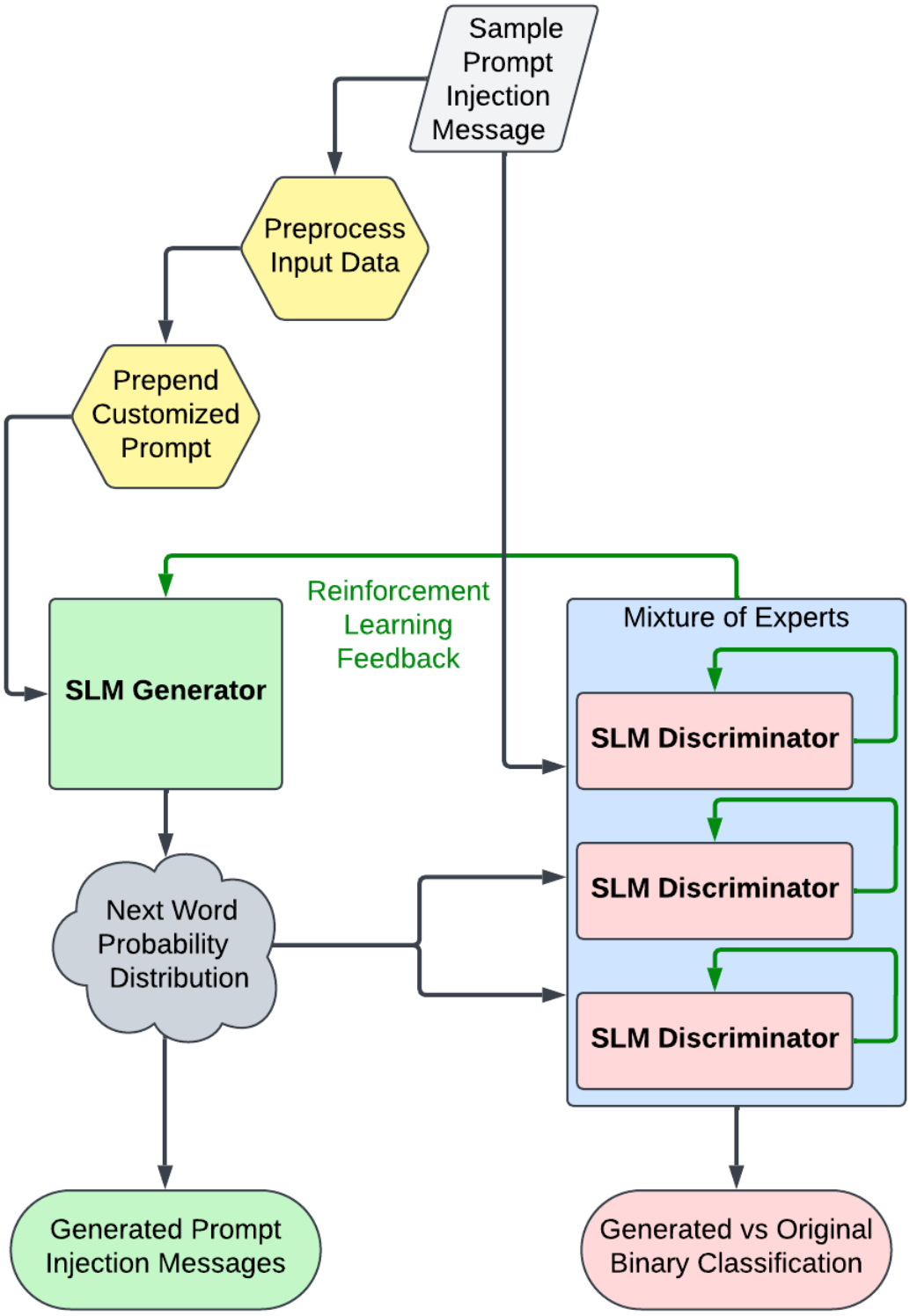

Figure 5.1: RLAIF Pipeline with Mixture of Expert Feedback

Chapter 6

Conclusion

In this thesis, we explored the combination of a fine-tuned SLM with a Reinforcement Learning with Artificial Intelligence Feedback pipeline. Leveraging GPT-2 Large as our SLM, we demonstrated that RLAIF could provide additional alignment to the ground truth distribution in a semantic manner, going beyond unigram or bigram matching. Furthermore, we addressed issues such as mode collapse and overfitting while ensuring training time remained reasonable. These findings underscore the importance of selecting an appropriate reference model and designing effective reward mechanisms.

Our work is significant because it demonstrates that smaller, more efficient models can be trained to generate adversarial prompt injection messages without relying on large-scale human feedback. By replacing Reinforcement Learning with Human Feedback (RLHF) with RLAIF, we introduce a scalable alternative that reduces reliance on slow and costly manual evaluation while maintaining model alignment. Additionally, our findings suggest that structured fine-tuning approaches, even on compact architectures, can effectively replicate attack behaviors observed in larger models. This has important implications for automated red teaming and model robustness evaluation, providing a lightweight yet effective method for stress-testing LLM defenses.

While the results suggest that RLAIF can enhance language model behavior for short

text generation, specifically in generating goal hijacking messages as a subset of prompt injection attacks, its broader applicability remains an open question. The improvements observed in this study leave open questions about scalability to larger models and real-world adversarial settings. Additionally, the reinforcement learning feedback loop, while no longer dependent on human annotators, still introduces computational overhead that must be optimized further.

Overall, this work provides a step toward more efficient adversarial text generation using small language models and AI-driven feedback loops. While the approach shows promise, further investigation is needed to fully assess its effectiveness in different contexts and refine its broader applicability to language model evaluation and security testing.

References

[1] Statista, *Natural Language Processing - Market Size*, 2025.

[2] A. Kumar, C. Agarwal, S. Srinivas, A. J. Li, S. Feizi, and H. Lakkaraju, "Certifying LLM Safety against Adversarial Prompting," Sep. 2023. URL: http://arxiv.org/abs/2309.02705.

[3] W. Zhang, X. Kong, C. Dewitt, T. Braunl, and J. B. Hong, "A Study on Prompt Injection Attack Against LLM-Integrated Mobile Robotic Systems," Aug. 2024. URL: http://arxiv.org/abs/2408.03515.

[4] K.-H. Hung, C.-Y. Ko, A. Rawat, I.-H. Chung, W. H. Hsu, and P.-Y. Chen, "Attention Tracker: Detecting Prompt Injection Attacks in LLMs," Nov. 2024. URL: http://arxiv.org/abs/2411.00348.

[5] B. Jiang, Y. Jing, T. Shen, T. Wu, Q. Yang, and D. Xiong, "Automated Progressive Red Teaming," Jul. 2024. URL: http://arxiv.org/abs/2407.03876.

[6] R. K. Sharma, V. Gupta, and D. Grossman, "Defending Language Models Against Image-Based Prompt Attacks via User-Provided Specifications," in *Proceedings - 45th IEEE Symposium on Security and Privacy Workshops, SPW 2024*, Institute of Electrical and Electronics Engineers Inc., 2024, pp. 112–131, ISBN: 9798350354874. DOI: 10.1109/SPW63631.2024.00017.

[7] R. Li, M. Chen, C. Hu, H. Chen, W. Xing, and M. Han, "GenTel-Safe: A Unified Benchmark and Shielding Framework for Defending Against Prompt Injection Attacks," Sep. 2024. URL: http://arxiv.org/abs/2409.19521.

[8] D. Pasquini, E. M. Kornaropoulos, and G. Ateniese, "Hacking Back the AI-Hacker: Prompt Injection as a Defense Against LLM-driven Cyberattacks," Oct. 2024. URL: http://arxiv.org/abs/2410.20911.

[9] D. Khomsky, N. Maloyan, and B. Nutfullin, "Prompt Injection Attacks in Defended Systems," Jun. 2024. URL: http://arxiv.org/abs/2406.14048.

[10] X. Suo, "Signed-prompt: A new approach to prevent prompt injection attacks against LLM-Integrated applications," 2024, p. 040 013. DOI: 10.1063/5.0222987. URL: https://pubs.aip.org/aip/acp/article-lookup/doi/10.1063/5.0222987.

[11] Lakera AI (https://www.lakera.ai), "mosscap_prompt_injection," 2023.

[12] J. Shi, Z. Yuan, Y. Liu, Y. Huang, P. Zhou, L. Sun, and N. Z. Gong, "Optimization-based Prompt Injection Attack to LLM-as-a-Judge," Mar. 2024. DOI: 10.1145/3658644.3690291. URL: http://arxiv.org/abs/2403.17710.

[13] F. Perez and I. Ribeiro, "Ignore Previous Prompt: Attack Techniques For Language Models," Nov. 2022. URL: http://arxiv.org/abs/2211.09527.

[14] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, "Not what you've signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection," Feb. 2023. URL: http://arxiv.org/abs/2302.12173.

[15] J. Ebrahimi, D. Lowd, and D. Dou, "On Adversarial Examples for Character-Level Neural Machine Translation," Tech. Rep., pp. 653–663. URL: https://www.perspectiveapi.com.

[16] X. Liu, Z. Yu, Y. Zhang, N. Zhang, and C. Xiao, "Automatic and Universal Prompt Injection Attacks against Large Language Models," Mar. 2024. URL: http://arxiv.org/abs/2403.04957.

[17] A. Zou, Z. Wang, N. Carlini, M. Nasr, J. Z. Kolter, and M. Fredrikson, "Universal and Transferable Adversarial Attacks on Aligned Language Models," Jul. 2023. URL: http://arxiv.org/abs/2307.15043.

[18] J. Ebrahimi, A. Rao, D. Lowd, and D. Dou, "HotFlip: White-Box Adversarial Examples for Text Classification," Dec. 2017. URL: http://arxiv.org/abs/1712.06751.

[19] J. Yu, Y. Shao, H. Miao, J. Shi, and X. Xing, "PROMPTFUZZ: Harnessing Fuzzing Techniques for Robust Testing of Prompt Injection in LLMs," Sep. 2024. URL: http://arxiv.org/abs/2409.14729.

[20] A. Salem, A. Paverd, and B. Köpf, "Maatphor: Automated Variant Analysis for Prompt Injection Attacks," Dec. 2023. URL: http://arxiv.org/abs/2312.11513.

[21] S. Toyer, O. Watkins, E. A. Mendes, *et al.*, "Tensor Trust: Interpretable Prompt Injection Attacks from an Online Game," Nov. 2023. URL: http://arxiv.org/abs/2311.01011.

[22] S. Abdali, R. Anarfi, C. Barberan, and J. He, "Securing Large Language Models: Threats, Vulnerabilities and Responsible Practices," Mar. 2024. URL: http://arxiv.org/abs/2403.12503.

[23] N. Stiennon, L. Ouyang, J. Wu, D. M. Ziegler, R. Lowe, C. Voss, A. Radford, D. Amodei, and P. Christiano, "Learning to summarize from human feedback," Sep. 2020. URL: http://arxiv.org/abs/2009.01325.

[24] L. Ouyang, J. Wu, X. Jiang, *et al.*, "Training language models to follow instructions with human feedback," Mar. 2022. URL: http://arxiv.org/abs/2203.02155.

[25] H. Lee, S. Phatale, H. Mansoor, *et al.*, "RLAIF vs. RLHF: Scaling Reinforcement Learning from Human Feedback with AI Feedback," Sep. 2023. URL: http://arxiv.org/abs/2309.00267.

[26] Y. Bai, S. Kadavath, S. Kundu, *et al.*, "Constitutional AI: Harmlessness from AI Feedback," Dec. 2022. URL: http://arxiv.org/abs/2212.08073.

[27] P. Christiano, J. Leike, T. B. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," Jun. 2017. URL: http://arxiv.org/abs/1706.03741.

[28] C.-Y. Lin, "ROUGE: A Package for Automatic Evaluation of Summaries," Tech. Rep.

[29] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," Aug. 2019. URL: http://arxiv.org/abs/1908.10084.

[30] L. McInnes, J. Healy, and J. Melville, "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction," Feb. 2018. URL: http://arxiv.org/abs/1802.03426.

[31] H. W. Chung, L. Hou, S. Longpre, *et al.*, "Scaling Instruction-Finetuned Language Models," Dec. 2022.

[32] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer," Tech. Rep., 2020, pp. 1–67. URL: http://jmlr.org/papers/v21/20-074.html..

[33] A. Reuther, J. Kepner, C. Byun, *et al.*, "Interactive Supercomputing on 40,000 Cores for Machine Learning and Data Analysis," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, IEEE, Sep. 2018, pp. 1–6, ISBN: 978-1-5386-5989-2. DOI: 10.1109/HPEC.2018.8547629.

[34] A. Q. Jiang, A. Sablayrolles, A. Mensch, *et al.*, "Mistral 7B," Oct. 2023. URL: http://arxiv.org/abs/2310.06825.

[35] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-Rank Adaptation of Large Language Models," Jun. 2021. URL: http://arxiv.org/abs/2106.09685.

[36] D. Han and M. Han, *Unsloth*, 2023.

[37] OpenAI, *Introducing Triton: Open-source GPU programming for neural networks*, 2021.

[38] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," Sep. 2014. URL: http://arxiv.org/abs/1409.3215.

[39] Y. J. Ma, W. Liang, G. Wang, D.-A. Huang, O. Bastani, D. Jayaraman, Y. Zhu, L. Fan, and A. Anandkumar, "Eureka: Human-Level Reward Design via Coding Large Language Models," Oct. 2023. URL: http://arxiv.org/abs/2310.12931.

[40] Y. Bai, A. Jones, K. Ndousse, *et al.*, "Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback," Apr. 2022. URL: http://arxiv.org/abs/2204.05862.

[41] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," Jul. 2017. URL: http://arxiv.org/abs/1707.06347.

[42] Z. Ze Yu, Z. Hui, L. Jia Jaw, and B. Kian Hsiang Low, "Fine-tuning Large Language Models with Generative Reward Modelling," Tech. Rep.

[43] J. Wu, X. Hu, Y. Wang, B. Pang, and R. Soricut, "Omni-SMoLA: Boosting Generalist Multimodal Models with Soft Mixture of Low-rank Experts," Dec. 2023. URL: http://arxiv.org/abs/2312.00968.

[44] C. Hardy, E. L. Merrer, and B. Sericola, "MD-GAN: Multi-Discriminator Generative Adversarial Networks for Distributed Datasets," Nov. 2018. DOI: 10.1109/IPDPS.2019.00095. URL: http://arxiv.org/abs/1811.03850%20http://dx.doi.org/10.1109/IPDPS.2019.00095.

[45] J. Duan, S. Zhang, Z. Wang, *et al.*, "Efficient Training of Large Language Models on Distributed Infrastructures: A Survey," Jul. 2024. URL: http://arxiv.org/abs/2407.20018.